

# Зміст

<b>1 Засоби розробки</b>	<b>5</b>
1.1 Основи JavaScript . . . . .	5
1.2 Інструментарій . . . . .	10
<b>2 Як працює доповнена реальність у веб-браузері</b>	<b>17</b>
2.1 Основи 3D-рендерінгу . . . . .	17
2.2 Відстеження та доповнена реальність . . . . .	24
<b>3 Відстеження зображень</b>	<b>29</b>
3.1 Підготовка зображення для MindAR . . . . .	29
3.2 Підміна веб-камери . . . . .	34
3.3 Застосування контейнеру для WebAR . . . . .	39
3.4 Завантаження текстур . . . . .	40
3.5 Завантаження моделей . . . . .	45
3.6 Відстеження декількох зображень . . . . .	54
3.7 Анімація моделей . . . . .	60
3.8 Опрацювання подій . . . . .	62
3.9 Взаємодія з користувачем . . . . .	65
3.10 Відтворення відео . . . . .	72
3.11 Видалення хромакею з відео . . . . .	74
3.12 CSS-рендеринг . . . . .	82
3.13 Відтворення потокового відео з Vimeo . . . . .	85
3.14 Відтворення потокового відео з YouTube . . . . .	88
3.15 Приклад: візитівка у доповненій реальності . . . . .	95
<b>4 Відстеження обличчя</b>	<b>107</b>
4.1 Опорні точки обличчя . . . . .	107
4.2 Проблема оклюзії . . . . .	110
4.3 Накладання маски на обличчя . . . . .	113
4.4 Перемикання камери . . . . .	117
4.5 Захоплення кадрів . . . . .	119
4.6 Застосування Web Share API на мобільних пристроях . . . . .	123
4.7 Приклад: примірка віртуальних аксесуарів . . . . .	128

<b>5 Відстеження довкілля</b>	<b>141</b>
5.1 Початок роботи із WebXR . . . . .	141
5.2 Застосування компоненту ARButton . . . . .	149
5.3 Управління контроллерами . . . . .	153
5.4 Розміщення об'єктів . . . . .	156
5.5 Перевірка дотику . . . . .	160
<b>6 Інтеграція машинного навчання</b>	<b>169</b>
6.1 Спільний доступ до відео з камери . . . . .	169
<b>7 Інші технології WebAR</b>	<b>176</b>
7.1 Вступ до A-Frame . . . . .	176
7.2 A-Frame та WebXR . . . . .	176
7.3 Вступ до model-viewer . . . . .	176
7.4 Вступ до комерційних SDK . . . . .	176
<b>8 Лабораторні роботи</b>	<b>181</b>
8.1 Перша . . . . .	181
<b>9 Віртуальна хімічна лабораторія у доповненій реальності: від ідеї до реалізації</b>	<b>187</b>
9.1 Ідея . . . . .	187
9.2 Підготовка маркерів, зображень та відео . . . . .	187
9.3 Програмна реалізація . . . . .	192
9.3.1 Початкова сторінка . . . . .	192
9.3.2 Реалізація віртуальної лабораторії . . . . .	192
9.3.3 Розгортання програмного забезпечення . . . . .	200
9.3.4 Тестування віртуальної лабораторії . . . . .	201
<b>А Файл index.html</b>	<b>201</b>
<b>Б Файл ar.html</b>	<b>202</b>

# Вступ

Розробка веб-додатків доповненої реальності (WebAR) відрізняється від інших способів розробки тим, що є крос-платформовою і не вимагає встановлення розроблених додатків, адже це просто звичайні веб-сторінки.

Наразі найвідоміша у світі бібліотека для розробки WebAR – AR.js (<https://ar-js-org.github.io/AR.js-Docs/>), проте XiyKim Юен (HiuKim Yuen, <https://www.youtube.com/channel/UC-JyA1Z1-p0wgxj5WEX56wg/featured>), один із її розробників, створив нову бібліотеку під назвою MindAR (<https://hiukim.github.io/mind-ar-js-doc/>) – більш компактну та технологічно розвинену. Це основна бібліотека, яку ми будемо використовувати для відстеження об'єктів та створення ефектів. Отримані знання, безумовно, можуть бути перенесені й на інші, зокрема, комерційні засоби.

WebXR – це JavaScript API для створення імерсивних ресурсів убраузерах: AR (augmented reality – доповненої реальності) та VR (virtual reality – віртуальної реальності). Будучи стандартом, він, ймовірно, стане однією з найбільш досконаліх веб-технологій 2020-х років. Це, безумовно, те, що ви повинні взяти на озброєння, якщо ви серйозно ставитеся до WebAR або WebVR.

У цьому курсі ми також дізнаємося, як за допомогою TensorFlow.js (<https://www.tensorflow.org/js>) інтегрувати моделі машинного навчання у WebAR додатки для створення високоінтерактивних і цікавих ефектів, наприклад, використання жестів рук або міміки для управління AR-контентом.

Ви не зможете розробити жодних серйозних AR додатків без опанування фреймворку для 3D-візуалізації, такого, як three.js (<https://threejs.org/>).

Розробка програмного забезпечення із доповненою реальністю часто вимагає зовнішнього пристрою для тестування. Полегшити цей процес можна за допомогою віддаленого налагодження та заздалегідь записаних відео, що імітують веб-камеру.

Важливою родзинкою цього курсу є те, що він допоможе зміцнити знання про те, як працює AR в середовищі браузера.

Посібник із курсу структурований у такий спосіб:

Розділ 1 допоможе налаштувати середовище розробки.

Розділ 2 є оглядом фундаментальних концепцій WebAR.

Розділи 3, 4 та 5 є основними для набуття досвіду розробки WebAR додатків з використанням MindAR та three.js.

У розділі 6 обговорюється інтеграція AR з моделями машинного навчання TensorFlow.js.

У розділі 7 коротко розглянуті інші важливі технології WebAR, в тому числі A-Frame (<https://aframe.io/>) та комерційні SDK.

# 1 Засоби розробки

## 1.1 Основи JavaScript

Для початку застосування JavaScript достатньо знати мінімальний синтаксис цієї мови:

1. *Змінні* у JavaScript створюються з літер, цифр, знаків долару та підкреслення:

а) при першому зверненні до них:

```
x = 1 // створити змінну x та надати їй значення 1
```

б) за допомогою ключового слова `var`

так:

```
var x // створити змінну x  
x = 1 // надати значення змінній
```

або так

```
var x = 1 // створити змінну x та надати їй значення 1
```

Змінні, створені без використання `var`, стають глобальними. Змінні також можуть бути оголошені за допомогою `let` (для змінної рівня блоку) та `const` (для сталої).

2. *Коментари* створюються аналогічно до C++:

```
// однорядковий коментар  
/*  
багаторядковий  
коментар  
*/
```

3. *Прості типи даних*:

- рядковий (`string`) – визначається подвійними або одинарними лапками і використовується для символічних даних;

- числовий (**number**) – визначається відсутністю лапок і використовується для дійсних чисел (наприклад, 345 – ціле десяткове, 34.5 – число з плаваючою точкою, 0b1011 – ціле двійкове, 0o377 – вісімкове, 0xFF – шістнадцяткове, **Infinity** –  $+\infty$ , **NaN** – помилкове число);
- логічний (**boolean**) – визначається відсутністю лапок і використовується для значень **true** = 1 або **false** = 0;
- символічний (**Symbol**) – тип унікального незмінного ідентифікатору;
- невизначений (**undefined**) – тип будь-якої неініціалізованої змінної (такої, якій не було надане значення).

4. *Спеціальні типи даних:*

- порожній (**null**) – відсутність даних у оголошенні змінній;
- об'єкт (**object**) – програмний об'єкт (посилання на нього);
- функція (**function**) – визначення функції.

5. *Основні оператори:*

- + додавання як бінарний, перетворення рядка на число як унарний;
- віднімання як бінарний, зміна знаку як унарний;
- \* множення;
- / ділення;
- % ділення за модулем (остача);
- ++ інкремент (збільшення на 1);
- = надання значення;
- += додати та надати значення;
- = відняти та надати значення;
- \*= помножити та надати значення;
- /= поділити та надати значення;
- %= знайти остаточу від ділення та надати значення;
- += додати та надати значення;
- == дорівнює;
- != не дорівнює;

>	більше;
>=	більше або рівний;
<	менше;
<=	менше або рівний;
====	ідентичний (дорівнює та одного типу);
!==	не ідентичний;
!	логічне заперечення;
	логічна диз'юнкція;
&&	логічна кон'юнкція;
&	побітова кон'юнкція;
	побітова диз'юнкція;
^	бінарна виключна диз'юнкція;
<<	побітовий зсув вліво;
>>	побітовий зсув вправо (із збереженням знаку);
>>>	побітовий зсув вправо (без збереження знаку);
~	побітове заперечення.

#### 6. Визначення функції:

```
function ім'я_функції(параметр1, параметр2, ..., параметрn)
{
    оператори
    return значення_що_повертається;
}
```

або

```
var ім'я_функції = function(параметр1, параметр2, ..., параметрn)
{
    оператори
    return значення_що_повертається;
}
```

або

```
var ім'я_функції = new Function('параметр1', 'параметр2', ..., 'параметрn',
    'оператори; return значення_що_повертається');
```

#### 7. Умовний вираз:

```
if(умова)
{
    оператори1
}
else // інакше, якщо умова не виконалась
{
    оператори2
}
```

або

```
результат = умова ? оператори1 : оператори2;
```

8. *Оператор вибору* надає можливість порівняти одну змінну з великою кількістю констант. Наприклад:

```
var a ;
switch ( a )
{
    case 1 : // якщо a = 1
        оператори
        [ break ; ]
    case 2 : // якщо a = 2
        оператори
        [ break ; ]
    default : // якщо a = 3
        оператори
        [ break ; ]
}
```

`case` порівнює змінну, зазначену в `switch` (змінна). `break` перериває виконання `case` або `default`, тобто якщо він буде відсутнім при виконанні хоча б одного `case`, виконаються всі наступні та `default`. `default` виконується, тільки якщо не виконається жоден із операторів `case`.

9. *Цикли*:

- `while` – цикл з передумовою, який триватиме до того моменту, коли умова не перестане виконуватись:

```
while(умова)
{
    оператори
}
```

- `do ... while` – цикл з післяумовою, який відрізняється від циклу `while` тим, що умова перевіряється наприкінці виконання блоку:

```
do {
    оператори
}while(умова)
```

- `for` – ітераційний цикл:

```
for(var змінна = початкове_значення; умова; крок циклу) {
    оператори
}
```

або

```
for (var ім'я_властивості in деякий_об'єкт) {
    // дії за допомогою деякий_об'єкт[ім'я_властивості];
}
```

10. Типи JavaScript поділяються на примітивні та об'єктні. Об'єкти можуть розглядатися як асоціативні масиви або хеші, тому часто реалізуються з використанням цих структур даних. *Стандартні об'єкти*: `Array`, `Date`, `Error`, `Math`, `Boolean`, `Function`, `Number`, `Object`, `RegExp`, `String`. Інші об'єкти – це “хост-об'єкти”, що визначаються не мовою, а середовищем виконання (наприклад, у браузері типові хост-об'єкти належать до DOM).

Об'єкти можуть бути створені за допомогою конструктора або літерала об'єкта (останнє є основою об'єктної нотації JavaScript – JSON):

```
var anObject = new Object(); // конструктор
// літерали
var objectA = {};
var objectB = {index1:'значення 1', index2:'значення 2'};
```

Як було показано далі, для доступу до даних та методів об'єкту застосовується оператор «точка» ( . ).

Все, що стосується синтаксису JavaScript/ECMAScript, можна знайти у багатьох джерелах – наприклад, якісних відеолекціях Дугласа Крокфорда (<https://youtu.be/playlist?list=PLEzQf147-uEpvTa1bHDN1xUL2k1HUMHJu>).

## 1.2 Інструментарій

Для розробки мовами HTML та JavaScript основними засобами розробки є простий текстовий редактор і веб-браузер (рис. 1.1), в якому можна відкрити звичайну веб-сторінку HTML, збережену локально.

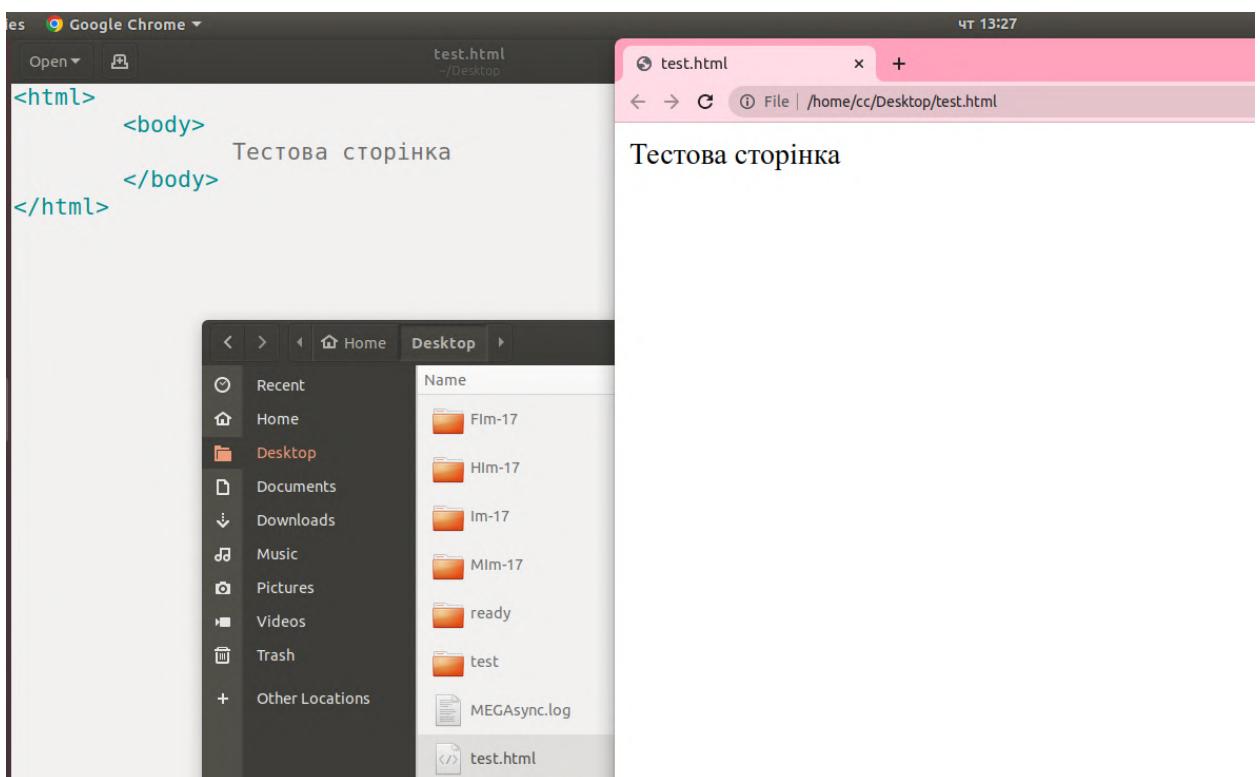


Рис. 1.1. Текстовий редактор – основний засіб розробки.

Однак це може не спрацювати для додатків, які потребують використання камери. Крім того, вам може знадобитися час від часу тестувати додатки на власних мобільних пристроях, тому краще встановити локальний веб-сервер (рис. 1.2).

Альтернативний спосіб – встановити розширення для Chrome, яке називається “Web Server for Chrome” (<https://chrome.google.com/webstore/>)

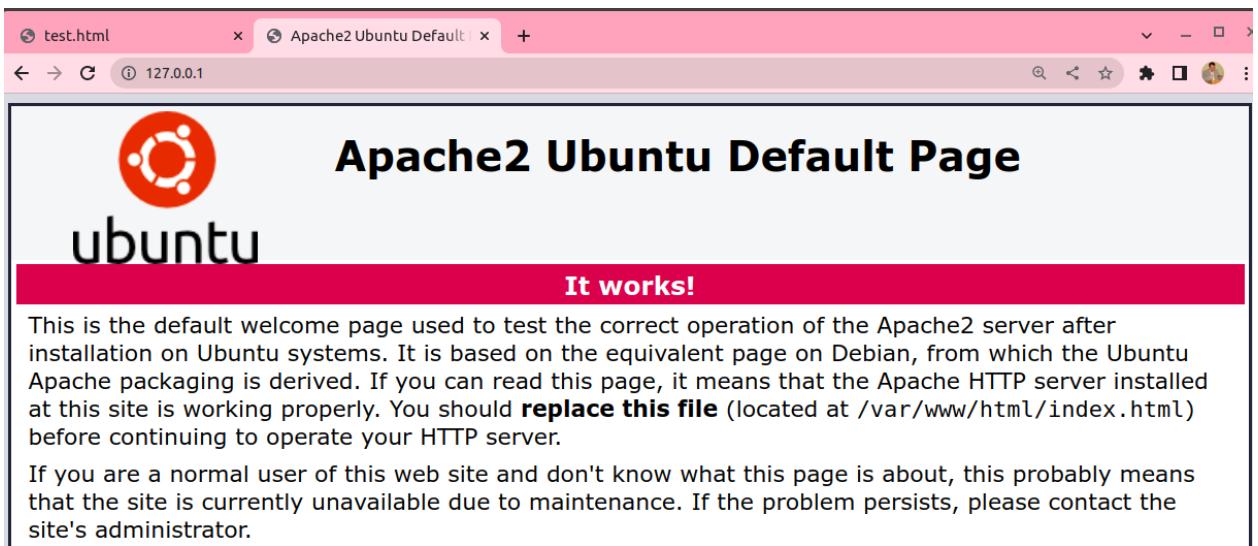


Рис. 1.2. Робота локального веб-серверу Apache.

detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb?hl=en) та є дуже простим у використанні. Після встановлення необхідно запустити сервер, а потім обрати кореневий каталог для веб-сторінок та обрати можливість доступу у локальній мережі, щоб інші пристрой також могли отримати доступ до веб-сторінки (рис. 1.3, 1.4); по завершенні роботи веб-сервер можна зупинити.

Корисним може бути вибір протоколу HTTPS у додаткових налаштуваннях – без його використання мобільний пристрій може не надати доступ до камери.

Технічно можна виконувати всю роботу з розробки та тестування безпосередньо на настільному браузері, але іноді все ж таки доцільно спробувати на мобільному телефоні (рис. 1.5).

Якщо пристрой підключені до однієї локальної мережі, у якій немає брандмауера, проблем із доступом до веб-серверу немає. Однак, якщо точка доступу до мережі знаходиться за брандмауером, можна використовувати ngrok (<https://ngrok.com/>) для того, щоб виконати перенаправлення трафіку з порту, доступ до якого обмежений.

Після встановлення ngrok та створення облікового запису на сайті <https://ngrok.com/> необхідно зареєструвати агент ngrok (<https://dashboard.ngrok.com/get-started/your-authtoken>) та запустити його, вказавши в якості параметру протокол (наприклад, http) та номер порту, доступ до якого закриває брандмауера (наприклад, 8887) (рис. 1.6).

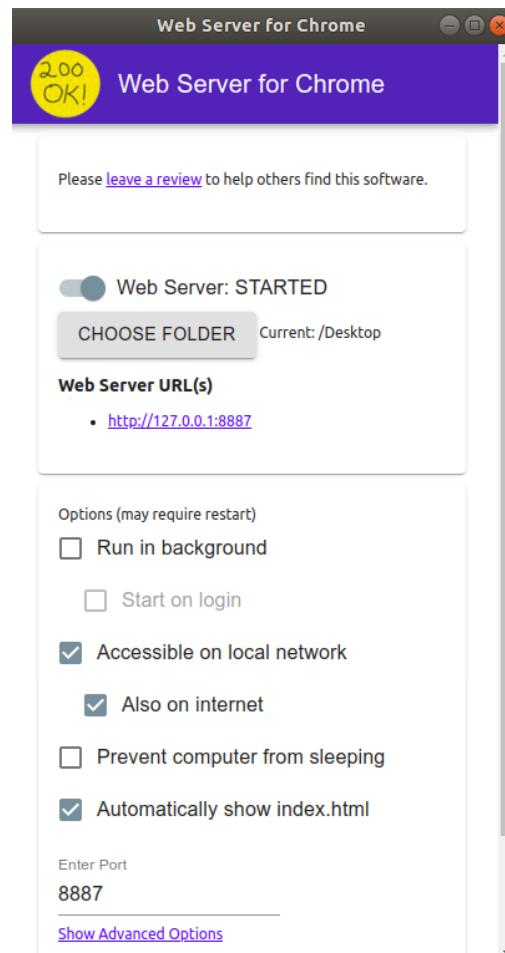


Рис. 1.3. Налаштування веб-серверу для Chrome.

Рис. 1.4. Робота веб-серверу для Chrome.

Після запуску ngrok надає посилання, яке глобально Інтернет за протоколом HTTPS – але лише у той час, коли працюють одночасно локальний веб-сервер та перенаправлення ngrok.



Рис. 1.5. Доступ до веб-сторінки у локальній мережі з різних пристройів.

Традиційно, налагодження веб-додатків передбачає перегляд консолі веб-браузера, куди виводяться повідомлення, що стосуються налагодження програми (рис. 1.7):

```
<html>
<body>
```

cc@cc-Latitude-5510: ~

```
File Edit View Search Terminal Help
cc@cc-Latitude-5510:~$ ngrok
Command 'ngrok' not found, but can be installed with:
sudo snap install ngrok
cc@cc-Latitude-5510:~$ sudo snap install ngrok
cc@cc-Latitude-5510:~$ ngrok config add-authtoken 2G4xHJoe5jroehknNm3skWnnImv_5RnkozQk
Auth token saved to configuration file: /home/cc/snap/ngrok/89/.config/ngrok/ngrok.yml
cc@cc-Latitude-5510:~$ ngrok http 8887
```

test.html | Web Server | Apps | Index of current directory |

← → ⌂ 🔍 b513-193-151-14-23.eu.ngrok.io/test.html

# Тестова сторінка

cc@cc-Latitude-5510: ~

```
File Edit View Search Terminal Help
ngrok (Ctrl+C to quit)
Try our new native Go library: https://github.com/ngrok/ngrok-go

Session Status          online
Account                 semerikov@gmail.com (Plan: Free)
Version                 3.1.0
Region                  Europe (eu)
Latency                 38ms
Web Interface           http://127.0.0.1:4040
Forwarding              https://b513-193-151-14-23.eu.ngrok.io -> http://localhost:8887

Connections             ttl     opn      rt1      rt5      p50      p90
                        3       0       0.04     0.01     0.01     0.01

HTTP Requests
-----
GET /test.html          200 OK
GET /                   200 OK
GET /favicon.ico        404 Not Found
```

Рис. 1.6. Застосування ngrok для обходу брандмауера.

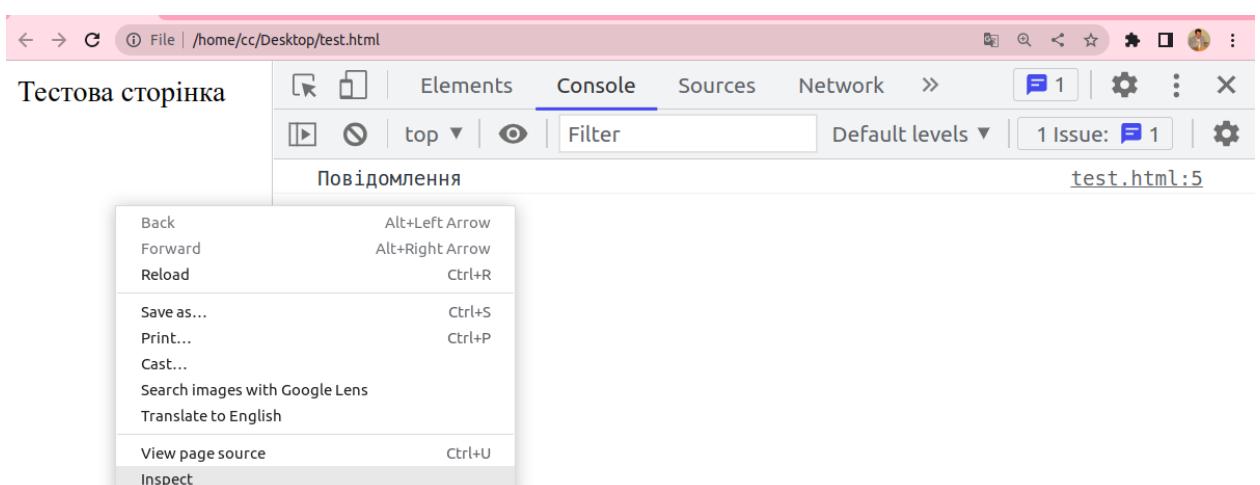


Рис. 1.7. Inspect – перегляд консолі веб-браузера.

```
Тестова сторінка
<script>
    console.log("Повідомлення");
</script>
</body>
</html>
```

Однак на мобільному пристрої це може бути не так просто. Тут допоможе RemoteJS (<https://remotejs.com/>) – натиснувши після переходу на сайт кнопку “Start Debugging”, отримаємо код агенту RemoteJS виду

```
<script
data-consolejs-channel="9817ec3e-a3f7-fbe3-3836-e2e2d07d5c99"
src="https://remotejs.com/agent/agent.js"></script>
```

Цей код необхідно скопіювати і вставити безпосередньо у веб-сторінку:

```
<html>
    <head>
        <script
            data-consolejs-channel="9817ec3e-a3f7-fbe3-3836-e2e2d07d5c99"
            src="https://remotejs.com/agent/agent.js">
        </script>
    </head>
    <body>
        Тестова сторінка
        <script>
            console.log("Повідомлення");
        </script>
    </body>
</html>
```

Після цього всі налагоджувальні повідомлення будуть надіслані на веб-сторінку з адресою [https://remotejs.com/viewer/agent\\_code](https://remotejs.com/viewer/agent_code), де agent\_code – значення змінної data-consolejs-channel (рис. 1.8).

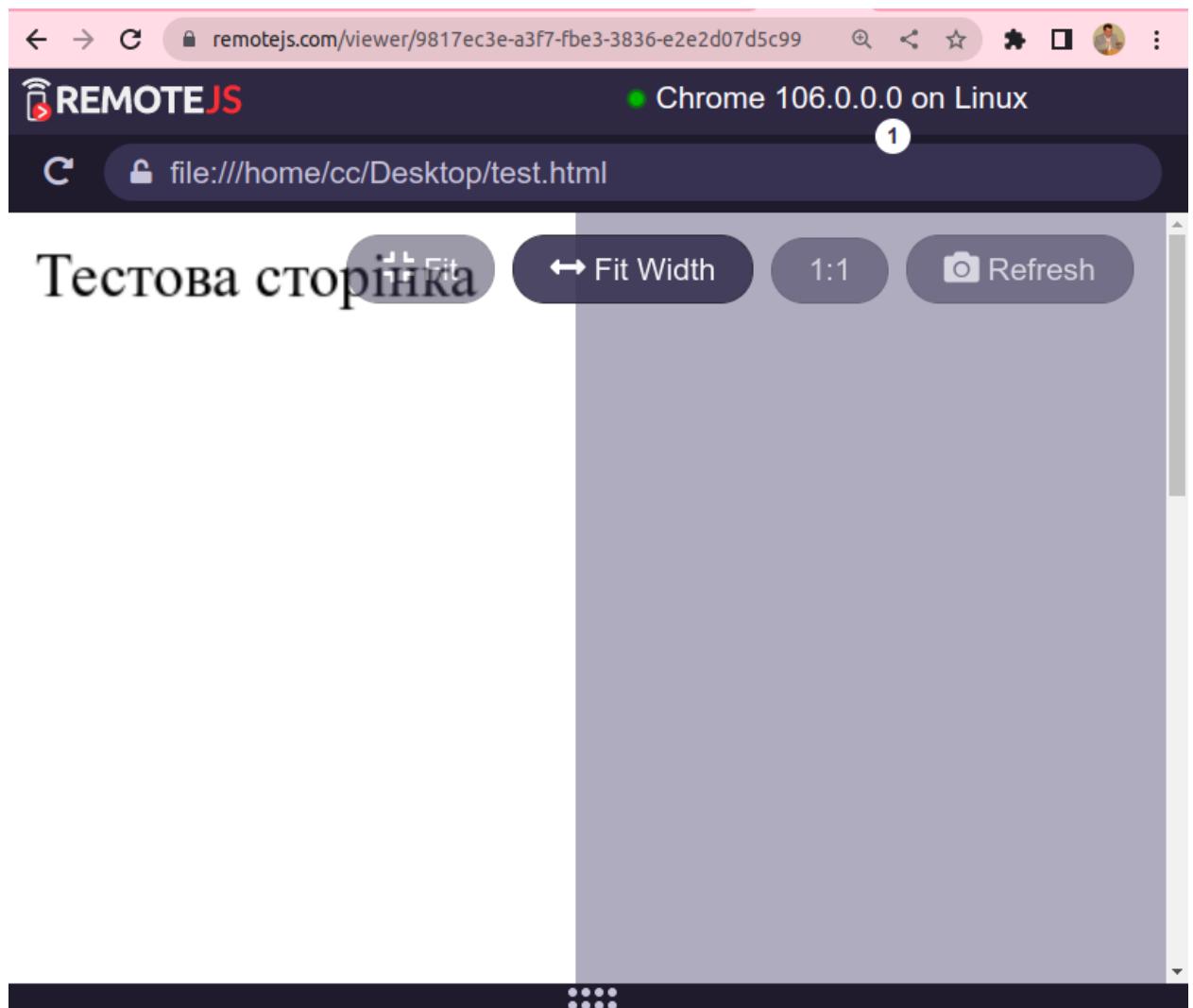


Рис. 1.8. Віддалене налагодження за допомогою агенту RemoteJS.

## 2 Як працює доповнена реальність у веб-браузері

### 2.1 Основи 3D-рендерінгу

WebGL ([https://developer.mozilla.org/en-US/docs/Web/API/WebGL\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API)) – JavaScript API для рендеринга 3D-графіки в браузерах. Він є крос-платформним стандартом відображення, який підтримують всі основні браузери. Проте низькорівневий код WebGL складно читати та писати, тому були створені більш зручні для користувача бібліотеки.

Three.js (<https://threejs.org/>) є однією з таких бібліотек, яку ми будемо використовувати. Її автор Рікардо Мігель Кабелло, також відомий як mrdoob, є одним із пionерів використання WebGL, тому ця бібліотека часто використовується при побудові інших бібліотек. Більшість WebAR SDK підтримують Three.js, тому вона дійсно потребує опанування для ефективної розробки веб-додатків з доповненою реальністю.

Щоб зрозуміти, як на високому рівні працює Three.js, доцільно провести аналогію з роботою фото- чи кінорежисеру, який:

- 1) налаштовує сцену шляхом розташування на ній об'єктів;
- 2) рухає камеру, щоб зафіксувати кадри з різних позицій та ракурсів (рис. 2.1).

Three.js не є спеціалізованою бібліотекою для доповненої реальності – вона містить суттєво більше функціональності, в тому числі тієї, що є більш придатною для веб-VR (освітлення, камери та ін.) (рис. 2.2).

Для створення додатку із застосуванням Three.js необхідно створити сторінку HTML, особливістю якою буде відсутність тіла, адже для генерації її вмісту буде використовуватись код JavaScript, що зазвичай розташовується між тегами `<script>` та `</script>`. Проте це найкраща практика, тому доцільним є розміщення коду в окремому файлі main.js:

```
<html>
<head>
<script src=".main.js" type="module"></script>
```



Рис. 2.1. Фото Alex Simpson ([https://unsplash.com/@m\\_simpsan](https://unsplash.com/@m_simpsan)).

```
</head>
<body>
</body>
</html>
```

Перше, що необхідно зробити, це імпортувати 3D бібліотеку. Наразі багато бібліотек, розміщених у хмарі, можна імпортувати безпосередньо через механізм CDN (content delivery network – мережу доставлення контенту), таких, як unpkg (<https://unpkg.com/>). На сторінці документації Three.js (<https://threejs.org/docs/index.html#manual/en/introduction/Installation>) подано відповідні приклади того, як це робити.

Так, при використанні звернення до CDN за шляхом <https://unpkg.com/three/> буде отриманий повний перелік версій Three.js, доступних через CDN. У випадку, коли необхідно зафіксувати номер версії бібліотеки (адже її функціональність змінюється – не лише з'являються нові можливості, а й зникають застарілі), доцільно чітко його вказати: наприклад, замість узагальненого <https://unpkg.com/three/build/three.module.js>

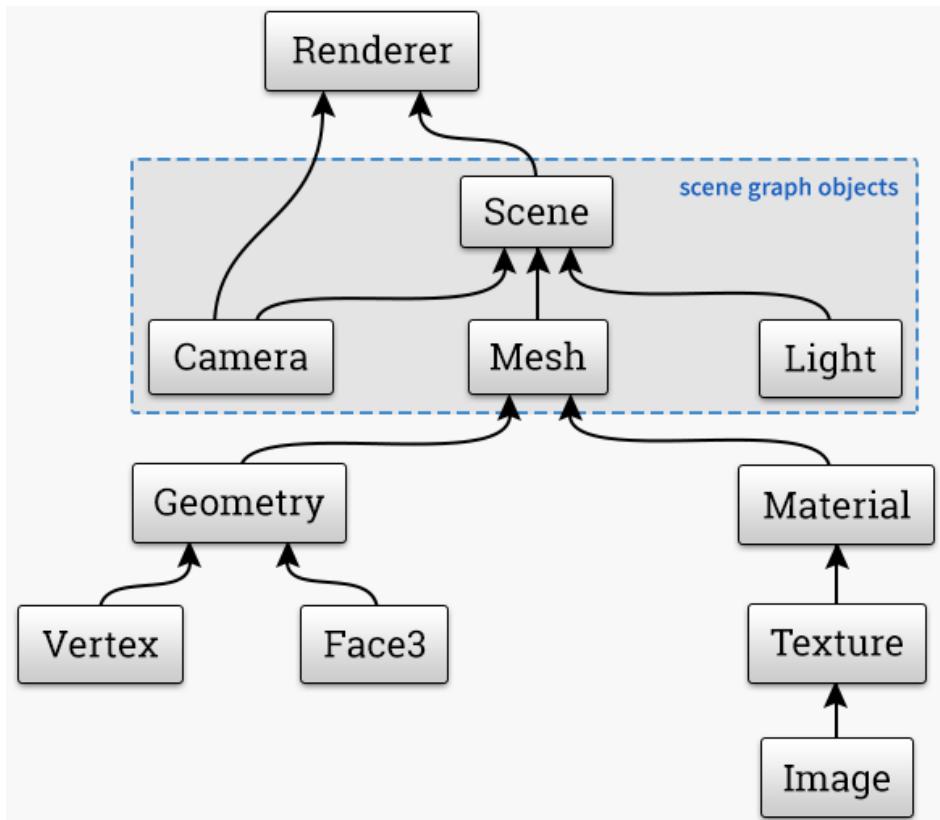


Рис. 2.2. Загальна структура Three.js.

вказати <https://unpkg.com/three@0.145.0/build/three.module.js>.

Імпорт із CDN

```
import * as THREE from "https://unpkg.com/three/build/three.module.js"
```

чи локального файлу

```
import * as THREE from "../three.js-r145/build/three.module.js";
```

вимагає додавання атрибуту `module` до тегу `script`:

```
<script src="./main.js" type="module"></script>
```

Намагання переглянути цей документ HTML локально у більшості сучасних браузерів завершиться невдачею через заборону локального (за протоколом `file://`) завантаження модулів, тому необхідним є застосування локального чи віддаленого веб-серверу.

Зазвичай код JavaScript має виконуватись після завершення завантаження документу HTML. Для того, щоб уберечити це, слід додати обробник події `DOMContentLoaded`, усередині якого розташувати код:

```

document.addEventListener("DOMContentLoaded", () => {
    const scene = new THREE.Scene();

    const geometry = new THREE.BoxGeometry(1, 1, 1);
    const material = new THREE.MeshBasicMaterial({color: "#0000FF"});
    const cube = new THREE.Mesh(geometry, material);
    cube.position.set(0, 0, -2);
    cube.rotation.set(0, Math.PI/4, 0);
    scene.add(cube);

    const camera = new THREE.PerspectiveCamera();
    camera.position.set(1, 1, 5);

    const renderer = new THREE.WebGLRenderer({alpha: true});
    renderer.setSize(500, 500);
    renderer.render(scene, camera);

    document.body.appendChild(renderer.domElement);
});

```

Як показано на рис. 2.2, основою є сцена – для її створення необхідно викликати конструктор без параметрів класу **Scene** (змінна **scene** – об'єкту класу **Scene** створюється динамічно за допомогою виклику **new**).

Створення об'єктів у Three.js відбувається у три кроки:

- 1) визначення геометрії об'єкту – векторів позиції, кольорів та ін.: так, **BoxGeometry** відповідає за прямокутний паралелепіпед;
- 2) визначення матеріалу – способу рендерингу об'єкту (його оптичні властивості – колір, фактура, бліск тощо): так, **MeshBasicMaterial** відповідає матеріалу, що має власний колір і не відбиває промені;
- 3) композиція геометрії та матеріалу виконується за допомогою **Mesh**.

Створений у такий спосіб куб буде розташований у початку координат. Для зміни його позиції скористаємося властивістю **position**, успадкованою

класом `Mesh` від свого батька – `Object3D`. Дано властивість є об’єктом класу `Vector3`, а `set` – його методом.

Аналогічно, властивість `rotation` зберігає кути нахилу об’єкту в радіанах. До речі, документація Three.js (див., наприклад, <https://threejs.org/docs/index.html#api/en/geometries/BoxGeometry>) містить інтерактивні демонстрації, що надають можливість переглянути різні об’єкти та модифікувати їх параметри. Рікардо Кабелло надає можливість конструювання сцени за допомогою візуального редактора за посиланням <https://threejs.org/editor/> – це може суттєво прискорити та полегшити процес її створення.

Всі об’єкти розміщаються на сцені за допомогою методу `add`.

Наступний об’єкт, що створюється – перспективна камера (`PerspectiveCamera`). Параметрами конструктора `PerspectiveCamera` є складові зрізаної піраміди огляду: перший – вертикальне поле зору (кут у градусах), другий – співвідношення сторін камери, третій – найближча площа, четвертий – найдальша (рис. 2.3).

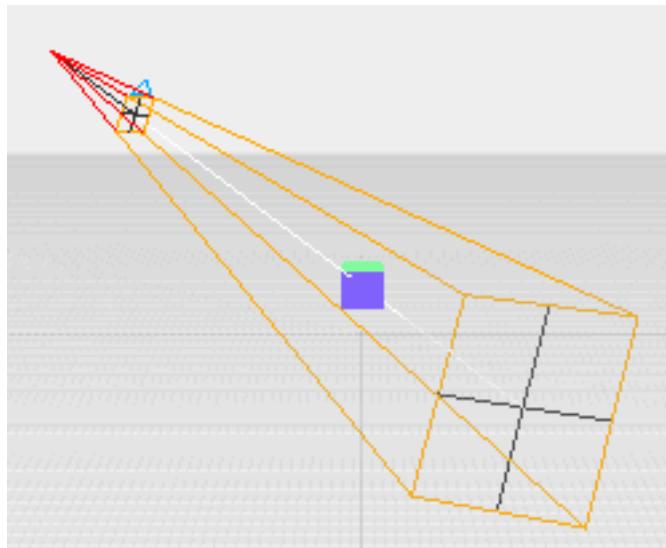


Рис. 2.3. Зрізана піраміда огляду.

Для зміни позиції камери скористаємося її властивістю `position`.

Рендерер – це те, що буде відображати 3D-модель на полотні з урахуванням матеріалу, текстури та освітлення. Змінна `renderer` створюється динамічно за допомогою виклику `new` як об’єкт класу `WebGLRenderer`. Одноіменна функція-конструктор класу в якості параметру приймає об’єкт у форматі JSON. Для роботи WebAR додатків важливо, щоб сцена була про-

зорою – тоді на неї можна буде накласти відеопотік з камери. Це досягається встановленням значенням параметру `alpha` у `true`.

Повний перелік властивостей та методів класу `WebGLRenderer` доступні у документації; в якості прикладу використано метод `setSize` встановлення висоти та ширини полотна (`canvas`) – площини, на яку проєціюється сцена.

Безпосередньо рендеринг виконує метод `render`, який відображає проекцію сцени на холст із точки зору камери.

Останній крок – зв’язування полотна зі сторінкою HTML – виконується викликом

```
document.body.appendChild(renderer.domElement);
```

Отже, ми використовуємо елемент `canvas` для відображення результатів рендерингу (рис. 2.4).

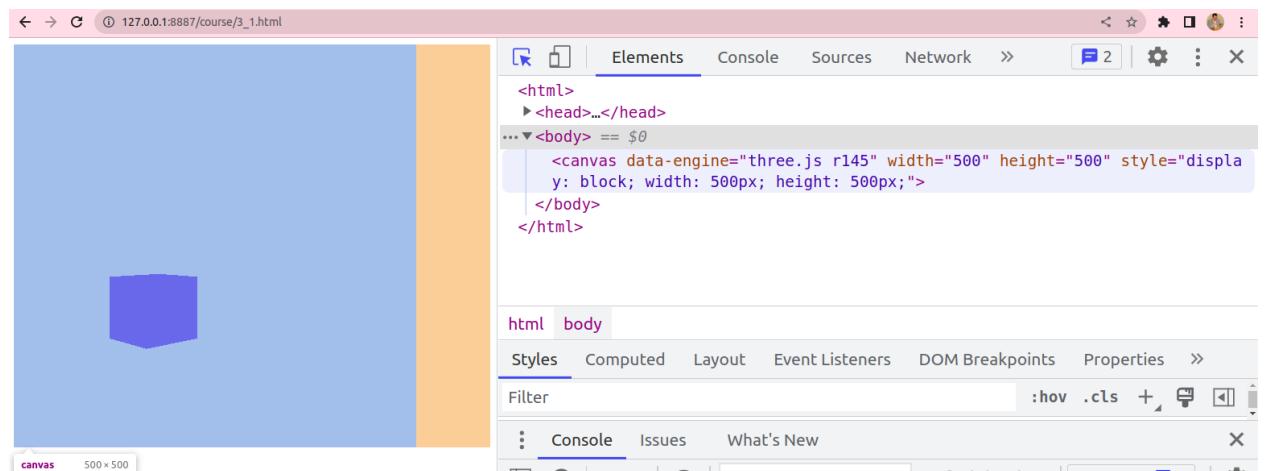


Рис. 2.4. Об’єкт `canvas`, убудований у документ HTML.

Перед зв’язуванням полотна зі сторінкою HTML для WebAR додатків необхідно виконати підключення відеопотоку. Для цього до документу HTML додамо тег (об’єкт) `video`:

```
const video = document.createElement("video");
```

Усередині об’єкту `video` розмістимо відеопотік з камери:

```
navigator.mediaDevices.getUserMedia({video:true})  
.then((stream) => {
```

```
    video.srcObject = stream;
    video.play();
});
```

`Navigator` є інтерфейсом доступу до стану та властивостей певного веб-браузера. Отримати відповідний об'єкт можна зверненням до властивості вікна `window.navigator`. Метод `mediaDevices` повертає посилання на об'єкт класу `MediaDevices`, що може бути використаний для отримання інформації про доступні медіапристрої (`enumerateDevices`), визначити їх властивості (`getSupportedConstraints`) та отримати доступ до них (`getUserMedia`).

Якщо виклик `getUserMedia` із запитом лише на відео (`video:true`) буде успішним, то буде отримано посилання на відеопотік `stream`, який необхідно пов'язати із властивістю `srcObject` об'єкту `video`. Останнім викликається метод `play` для початку відтворення (програвання) відеопотоку з камери у об'єкті `video`.

Налаштуюмо елементи каскадної таблиці стилів (CSS) для об'єктів `video` та `renderer`:

```
video.style.position = "absolute";
video.style.width = renderer.domElement.width;
video.style.height = renderer.domElement.height;
renderer.domElement.style.position = "absolute";
```

Значення параметру `position` встановлюється у `absolute` – такий елемент “щезає” з того місця, де він мав бути й позиціонується заново. Усі інші елементи розташовуються так, нібіто цього елементу ніколи не було, а ширина елементу встановлюється за його вмістом. У нашому випадку ширина та висота об'єкту `video` встановлені у аналогічні значення по-голотна, на якому працює `renderer`, тому застосування `position:absolute` як до `video`, так і до `renderer` надає можливість сумістити (накласти) ці два об'єкти.

Додати створений та налаштований об'єкт `video` до документу необхідно до додавання `renderer.domElement` – тоді зображення куба буде поверх відеопотоку:

```
document.body.appendChild(video);
```

На рис. 2.5 показано першу реалізацію WebAR, в якій реальний об'єкт з камери доповнений віртуальним об'єктом.



Рис. 2.5. Результат накладання.

Розміщення полотна поверх відео є основою WebAR. Єдине, чого бра-кує, це відображення об'єкту у більш доцільному місці та оновлення його положення відповідно до сигналу з камери, тобто відстеження об'єкту.

## 2.2 Відстеження та доповнена реальність

Змінити положення зображення можна шляхом переміщення віртуальної камери, змінюючи її позицію (координати) та нахил. Доцільні зміни вимагають відстеження об'єктів, тому поширилося класифікація доповненої реальності на маркерну, безмаркерну, координатну тощо.

Автор бібліотеки MindAR пропонує класифікацію доповненої реальності за типом відстеження.

Перший тип – *відстеження зображень*: у цьому типі віртуальні об'єкти з'являються поверх цільових зображень, які можуть бути маркерними (barcode-like, рис. 2.6), які мають заздалегідь визначену структуру, та природними (рис. 2.7), які можуть бути чим завгодно.

Зображення не обов'язково маю бути друкованими чи екранними – можуть бути навіть футболки з доповненою реальністю<sup>1</sup>.

Другий тип доповненої реальності – *відстеження обличчя*, за якого об'єкти прикріплюються до людського обличчя. Прикладами є фільтри в

<sup>1</sup>Див., наприклад, “9 ideas for creating tech-infused augmented reality T-shirts” від Ainars Klavins за посиланням <https://overlyapp.com/blog/9-ideas-for-creating-tech-infused-augmented-reality-t-shirts/>

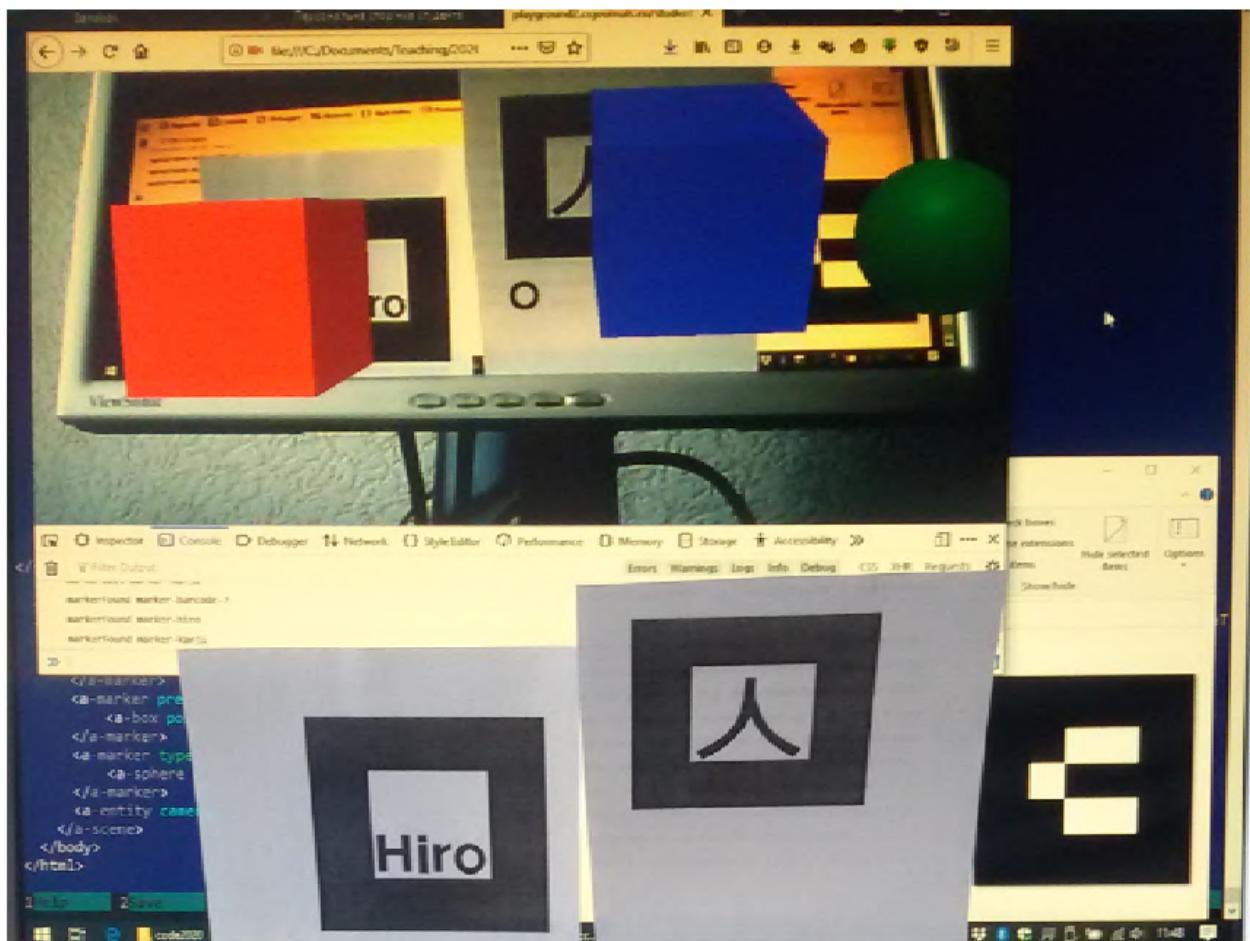


Рис. 2.6. Застосування маркерів.

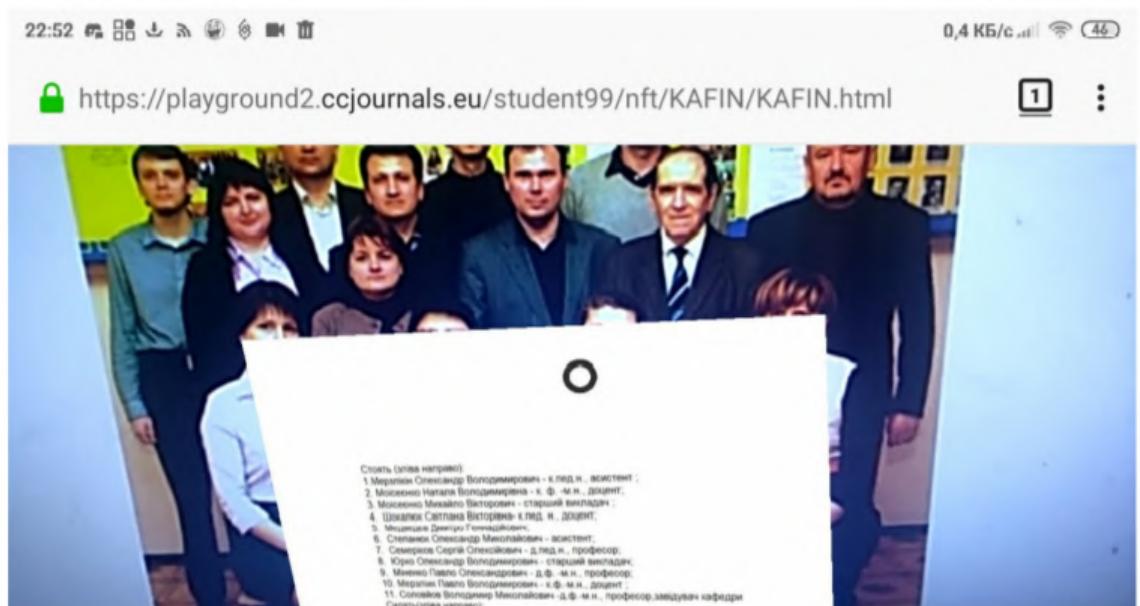


Рис. 2.7. Застосування природних зображень.

Instagram, Google Meet (рис. 2.8), кампанії у соціальних мережах, додатки для примірки віртуальних аксесуарів тощо.

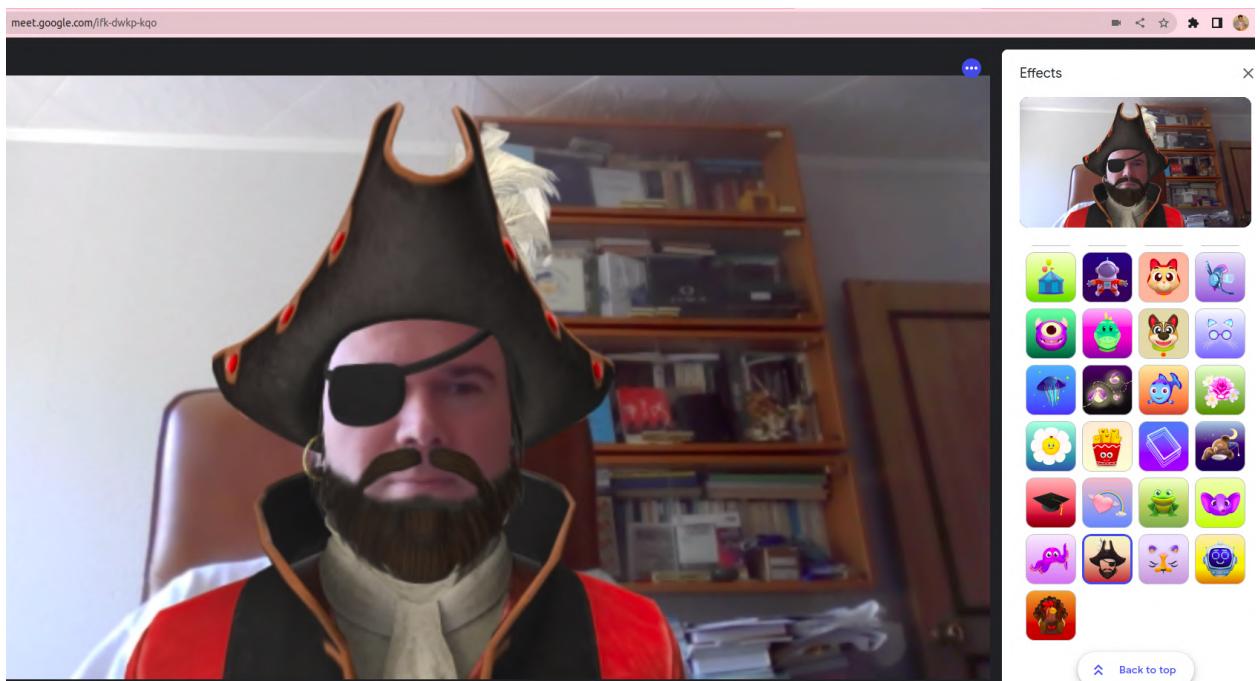


Рис. 2.8. Застосування фільтру для обличчя.

Третій тип доповненої реальності – *відстеження довкілля* (world tracking), який також називають безмаркерною доповненою реальністю. За такого типу відстеження об’єкти доповненої реальності можуть бути розміщений де завгодно, не обмежуючись конкретним зображенням, обличчям або фізичними об’єктами.

Додатки відстеження довкілля безперервно фіксують і відстежують навколоїшнє середовище і оцінюю фізичне положення користувача додатку. Найчастіше об’єкти доповненої реальності прикріплюються до певної поверхні (рис. 2.9), зокрема, до землі.

*Геокоординатна доповнена реальність* (location-based AR), відома за Pokémon GO, Ingress тощо (рис. 2.10) передбачає прив’язку контенту до певного географічного положення – широти та довготи. Зазвичай ці програми відстежують довкілля, оскільки доповнений вміст, як правило, прикріплений до землі, а геокоординатна частина є скоріше додатковою умовою, виконання якої приводить до початку відстеження довкілля (або обличчя) у певному місці.

Можуть бути визначені й інші типи відстеження – відстеження 3D-об’єктів, відстеження рук та ін.

Незважаючи на різноманіття бібліотек для доповненої реальності, їх основною задачею є визначення позиції віртуальної камери відповідно до

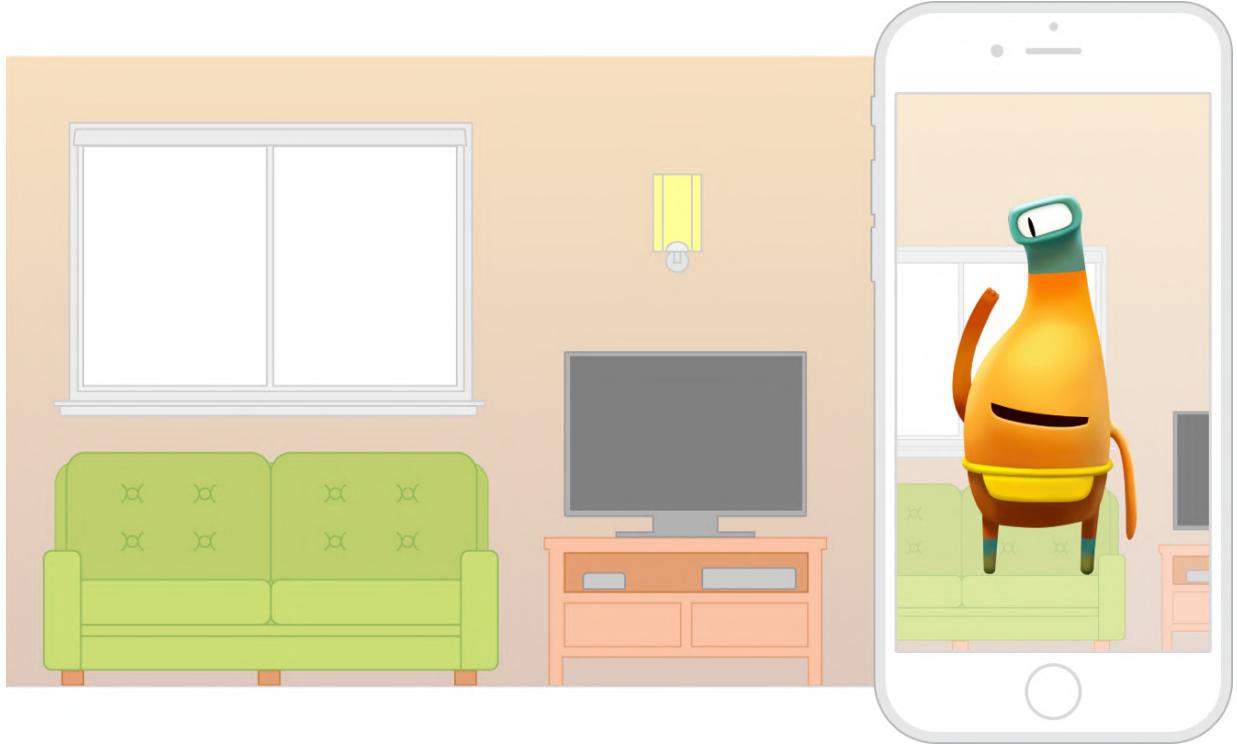


Рис. 2.9. Відстеження довкілля ([https://developer.apple.com, Documentation / ARKit / Configuration Objects / Understanding World Tracking](https://developer.apple.com/documentation/arkit/configuration_objects/understanding_world_tracking)).

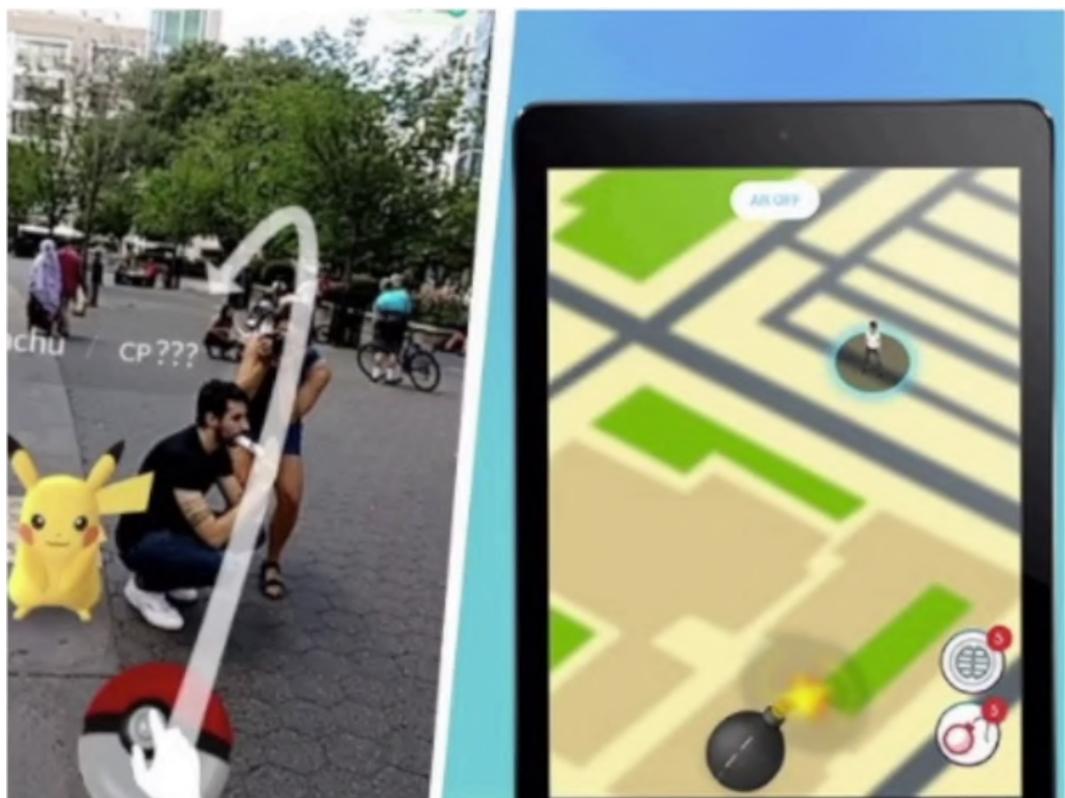


Рис. 2.10. Геокоординатна доповнена реальність.

відстежуваного об'єкту, що ілюструється наступним псевдокодом:

```
const ar = new SOME_AR_ENGINE();
while(true)
{
    await nextVideoFrameReady();
    const {position, rotation} = ar.computeCameraPose(video);
    camera.position = position;
    camera.rotation = rotation;
}
```

Спочатку необхідно ініціювати бібліотеку – певний AR-рушій, та отримати. Далі у безперервному циклі дочекатись кадр з відеопотоку реальної камери, визначити її положення (координати на нахил) та перемістити віртуальну камеру на полотні у те саме положення.

Нерідко, однак, рухають не віртуальну камеру, а об'єкти на сцені – тоді визначається положення не реальної камери, а відстежуваного об'єкту, після чого об'єкт віртуальної реальності переміщується у те саме положення, що й відстежуваний об'єкт:

```
const ar = new SOME_AR_ENGINE();
while(true)
{
    await nextVideoFrameReady();
    const {position, rotation} = ar.computeObjectPose(video);
    cube.position = position;
    cube.rotation = rotation;
}
```

### 3 Відстеження зображень

#### 3.1 Підготовка зображення для MindAR

Зображення, що відстежується, може бути будь-якого походження, проте воно має бути підготовленим: так, якщо воно містить зайві елементи (рис. 3.1, праворуч), їх необхідно видалити (рис. 3.1, ліворуч).

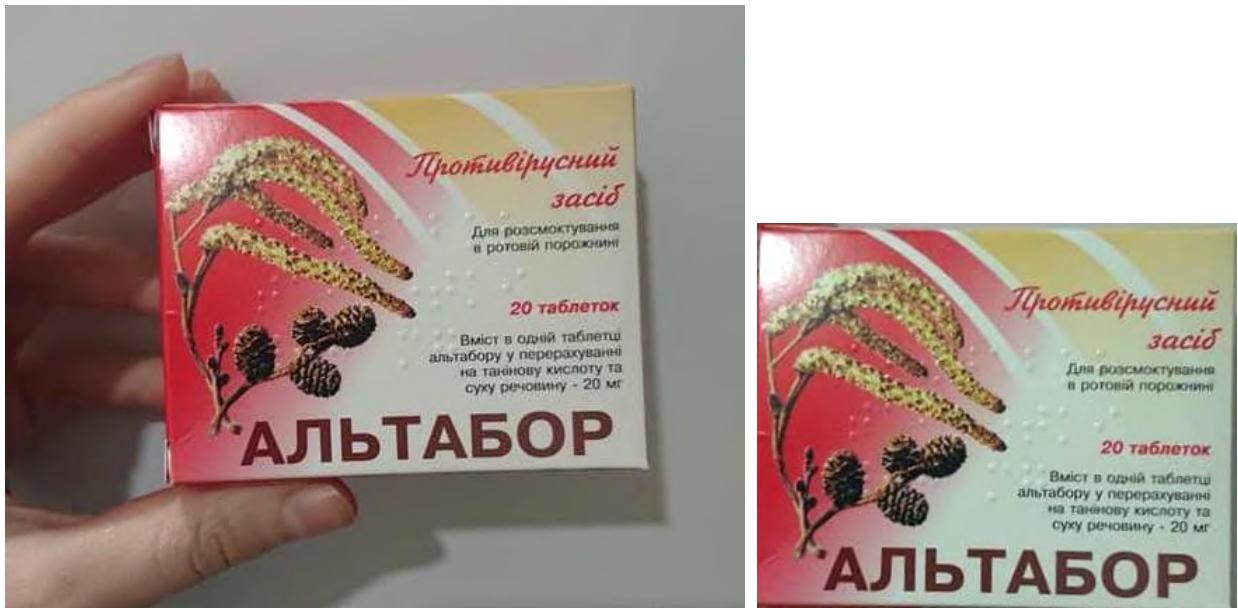


Рис. 3.1. Вихідне та підготовлене зображення.

Для розпізнавання зображення з використанням бібліотеки MindAR необхідно виділити на ньому опорні точки – елементи, за якими буде виконуватись розпізнавання. Це можна зробити за допомогою компілятора зображень, розміщеного за посиланням <https://hiukim.github.io/mind-ar-js-doc/tools/compile>.

Результатом роботи компілятора є бінарний файл `targets.mind`, що містить опис опорних точок (рис. 3.2), відстеження яких відбудуватиметься.

Інші бібліотеки мають схожі засоби отримання опису зображень, які часто називають компіляторами NFT маркерів (від natural feature tracking – відстеження природних зображень). Таке зображення повинно бути візуально складним та мати високу роздільність (тут деталі мають значення). Візуально складне зображення надає програмному забезпеченню багато можливостей для відстеження унікальних частин зображення, що легко розпізнаються.



Рис. 3.2. Візуалізація опорних точок.

Від фізичного розміру NFT маркера також залежить якість його розпізнавання – до малих за розміром зображень мобільний пристрій необхідно наблизити, у той час як від великих навпаки – тримати подалі.

Якість розпізнавання також залежить від освітленості екрану мобільного пристрою; крім того, камери з низькою роздільною здатністю зазвичай працюють краще, коли вони знаходяться близько до маркерів.

Після завантаження файлу `targets.mind` перейменуємо його на `altabor.mind` та створимо мінімальний файл HTML:

```
<html>
  <head>
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0">
    <script src=
      "https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
      ></script>
    <script src=".main.js"></script>
    <style>
      html, body {
```

```

        position: relative;
        margin: 0;
        width: 100%;
        height: 100%;
        overflow: hidden
    }
</style>
</head>
<body>
</body>
</html>

```

Згідно рекомендацій у документації MindAR, останню версію бібліотеки MindAR для спільноговідстеження зображень і роботи із Three.js (`mindar-image-three`) можна отримати з CDN за універсальним посиланням <https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js> або вказавши номер версії бібліотеки: <https://cdn.jsdelivr.net/npm/mind-ar@1.1.5/dist/mindar-image-three.prod.js>.

Стиль документу налаштований на повновіконне зображення. Так само, як і в попередньому прикладі, файл `main.js` міститиме мінімально необхідний для роботи код JavaScript:

```

const THREE = window.MINDAR.IMAGE.THREE;

document.addEventListener("DOMContentLoaded", () => {

});

```

Бібліотека Three.js є частиною MindAR (модуль `MINDAR.IMAGE.THREE` прикріплений до об'єкта `window`), тому для доступу до неї створюється синонім `THREE`. Наступний код розміщується усередині обробника події `DOMContentLoaded`:

```
const start = async() => {
```

```
}

start();
```

Подія `DOMContentLoaded` опрацьовується тоді, коли завершено завантаження документу HTML. Проте певний час може бути необхідний для того, щоб налаштувати веб-камеру та підготувати її до роботи. Для того, щоб дочекатись результатів певної функції, перед нею ставиться ключове слово `await`, яке не можна використати у звичайних функціях, тому у середині обробника створена та викликається асинхронна функція `start` із наступним вмістом:

```
const mindarThree = new window.MINDAR.IMAGE.MindARThree({
    container: document.body,
    imageTargetSrc: "altabor.mind",
});

const {renderer, scene, camera} = mindarThree;

const geometry = new THREE.PlaneGeometry(1, 1);
const material = new THREE.MeshBasicMaterial({
    color: 0x00ffff, transparent: true, opacity: 0.5
});
const plane = new THREE.Mesh(geometry, material);

const anchor = mindarThree.addAnchor(0);
anchor.group.add(plane);

await mindarThree.start();
renderer.setAnimationLoop(() => {
    renderer.render(scene, camera);
});
```

`mindarThree` є об'єктом класу `MindARThree`, що має два параметри: `container` – місце, асоційоване з відеопотоком (у нашому випадку це тіло документа HTML), та `imageTargetSrc` – шлях до файлу опису опорних точок відстежуваного зображення (результат роботи компілятора зображень з розширенням `.mind`).

Конструктор класу `MindARThree` має також три додаткові параметри, `uiError`, `uiScanning` та `uiLoading` – встановлення їх значень у `no` надасть можливість відключити початковий екран (подробиці – у документації: <https://hiukim.github.io/mind-ar-js-doc/examples/custom-ui>).

`MindARThree` також створює необхідні для роботи із `Three.js` об'єкти рендерер, сцену та камеру, доступні відповідно як поля `renderer`, `scene` і `camera` об'єкту `mindarThree`.

Далі створюється об'єкт `plane`, що визначається геометрією `PlaneGeometry` (площина одиничного розміру) та матеріалом із власним кольором `MeshBasicMaterial` (напівпрозорий ціанового кольору).

`anchor` – якірний об'єкт, що повертається викликом методу `addAnchor`, параметр якого відповідає номеру зображення, що розпізнається. Через те, що опрацьовується лише одне зображення, використовується його номер (0 – перше зображення, 1 – друге і т. д.). Якірні об'єкти використовуються для відстеження цільових зображень та надають позицію, в якій повинен бути розміщений об'єкт.

Замість того, щоб додавати площину безпосередньо до сцени, вона додається до складової якоря – об'єкту `group` класу `THREE.Group`, що визначає множину пов'язаних об'єктів, положенням, орієнтацією та видимістю яких можна керувати спільно. Ця якірна група управляється бібліотекою, яка буде постійно оновлюватимемо положення і орієнтацію групи відповідно до нашого набору для відстеження.

Метод `start` об'єкту `mindarThree` і є тим, для якого довелося зробити однойменну зовнішню функцію асинхронною. Це метод не лише виконує налаштування параметрів та вмикання камери (цього можна було й не чекати), а й завантажує у пам'ять веб-браузера усі необхідні дані (у нашему випадку – файл "altabor.mind").

Для того, щоб рендерер, камера та сцена запрацювали, необхідно створити функцію для їх візуалізації. У безіменній функції зворотного виклику, що створюється функцією `setAnimationLoop`, для кожного кадру із об'єкту `renderer` викликається метод `render`, параметрами якого є об'єкти `scene` та `camera` – це і є анімація на полотні.

У результаті отримаємо повнофункціональний WebAR додаток, що відстежує одне зображення (рис. 3.3).

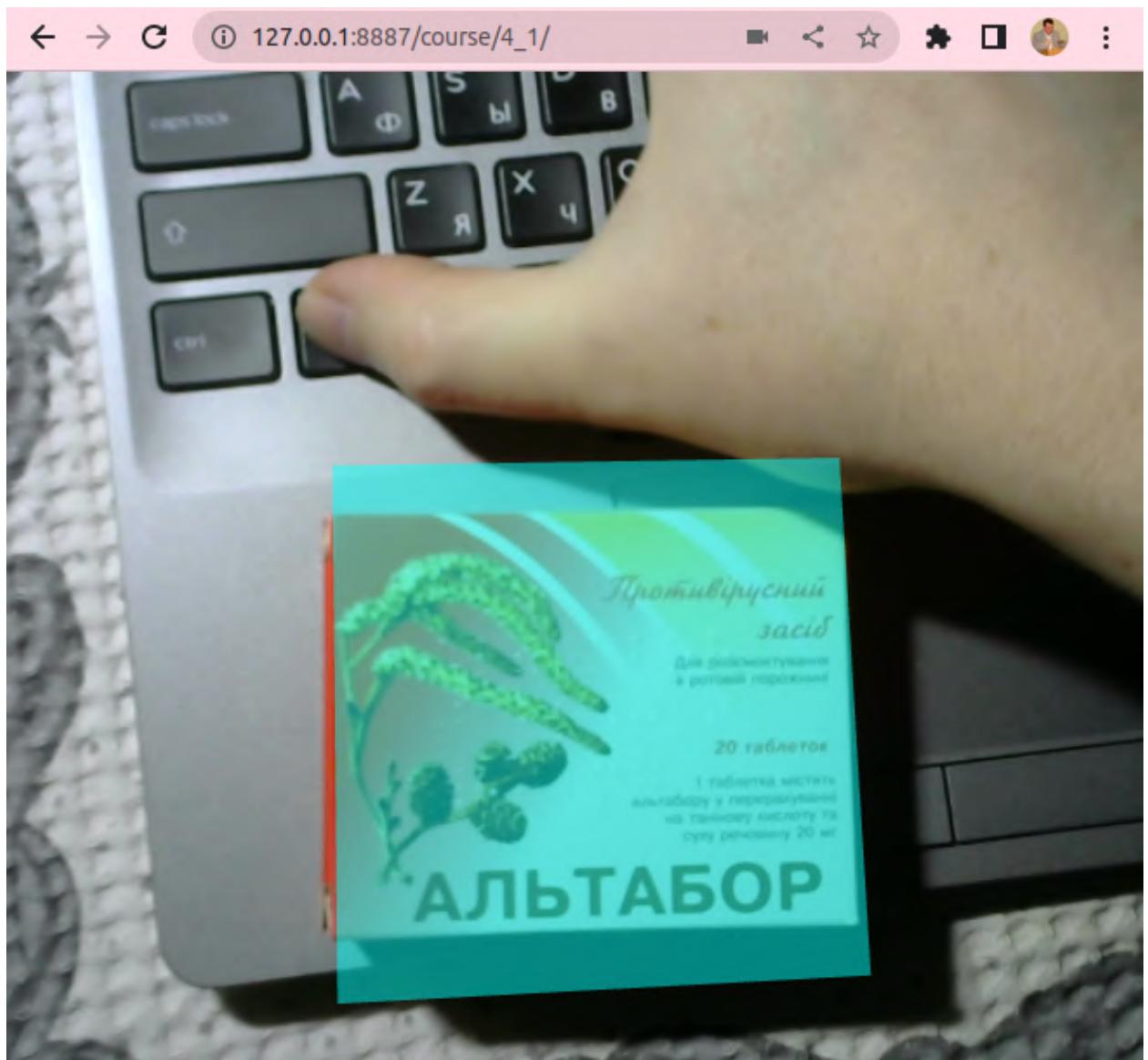


Рис. 3.3. Результат розпізнавання зображення.

Незважаючи на те, що відстежуване зображення є прямокутником, площа масштабувалась до нього за більшою стороною, тому вибір її розмірів як одиночної є цілком віправданим. Для повного перекривання реального зображення віртуальним об'єктом необхідно встановити його розміри пропорційно до розмірів реального: найбільший розмір (у нашому випадку – ширина) в 1, а найменший – у число між 0 та 1 (у нашому випадку – висота: 0.8).

## 3.2 Підміна веб-камери

Однією з проблем при розробці WebAR додатків є те, що кожного разу, коли додаток тестиється, необхідно вмикати камеру пристрою. Далеко

не завжди умови роботи (зокрема, освітлення, роздільна здатність камери тощо) сприяють тому, що тестування буде успішним, тому доцільним є використання наперед записаних відео, таких, як <https://github.com/ssemerikov/webar/blob/main/mock-video.mp4>.

Для цього потрібно підмінити API для запуску камери.

У розділі ми застосовували метод `navigator.mediaDevices.getUserMedia`, що запитує у користувача дозвіл на використання мультимедійного входу, який створює потік `MediaStream` з доріжками, що містять запитувані типи мультимедійних даних.

Цей потік може включати, наприклад, відеодоріжку (вироблену апаратним або віртуальним джерелом відео, таким як камера, пристрій відеозапису, служба демонстрації екраном тощо), аудіодоріжку (аналогічно, вироблену фізичним або віртуальним джерелом звуку, таким як мікрофон, АЦП тощо), і, можливо, інші типи доріжок.

Метод повертає об'єкт класу `Promise` (їх часто так й називають – проміси), яка перетворюється на об'єкт `MediaStream`. Якщо користувач відмовляє в дозволі або відповідний носій недоступний, то проміс відхиляється з генеруванням виключеної ситуації.

Відповідно до цього опису й перевизначимо функцію `getUserMedia` до виклику конструктору класу `MINDAR.IMAGE.MindARThree`:

```
navigator.mediaDevices.getUserMedia = () => {
  return new Promise((resolve, reject) => {
    const video = document.createElement("video");
    video.setAttribute("src", "mock-video.mp4");
    video.setAttribute("loop", "");

    video.oncanplay() => {
      video.play();
      resolve(video.captureStream());
    }
  });
}
```

У промісі, що повертається `getUserMedia`, створюється елемент `video`, для якого викликами `setAttribute` задаються атрибути `src` (джерело

відео – записаний відеофайл) та `loop` (циклічно повторювати відео). Метод `oncanplay` спрацьовує, коли відео готове до програвання (надходить подія `canplay`) – тоді метод `play` намагається розпочати програвання відео, а метод `captureStream` повертає `CanvasCaptureMediaStream` – відеозахоплення на полотні.

Як часто використовуваний, код для підміни потоку з камери доцільно оформити у бібліотеку `camera-mock.js`:

```
export const mockWithVideo = (path) => {
  navigator.mediaDevices.getUserMedia = () => {
    return new Promise((resolve, reject) => {
      const video = document.createElement("video");

      video.oncanplay = () => {
        const startButton = document.createElement("button");
        startButton.innerHTML = "start";
        startButton.style.position = 'fixed';
        startButton.style.zIndex = 10000;
        document.body.appendChild(startButton);

        startButton.addEventListener("click", () => {
          const stream = video.captureStream();
          video.play();
          document.body.removeChild(startButton);
          resolve(stream);
        });
      };
      video.setAttribute("loop", "");
      video.setAttribute("src", path);
    });
  };
}
```

Метод `mockWithVideo`, визначений у бібліотеці `camera-mock.js`, відрізняється від попереднього коду тим, що відео запускається не автоматично, а по натисканню кнопки, створеної за допомогою `createElement`,

що має ім'я `start` (значення властивості `innerHTML`), фіксоване місце (властивість `style.position`) та знаходиться поверх усіх елементів (забезпечується великим значенням `style.zIndex`).

По натисканню кнопки (подія `click`), крім захоплення відеопотоку, запуску програвання відео та повернення захопленого відеопотоку, відбувається ще видалення кнопки викликом `removeChild`.

Для використання методу `mockWithVideo` на початку файлу `main.js` його необхідно імпортувати з бібліотеки `camera-mock.js` командою

```
import {mockWithVideo} from "./camera-mock.js";
```

Для того, щоб команда імпорту спрацювала, необхідно змінити спосіб завантаження файлу `main.js` у документі HTML, додавши до тегу `script` параметр `type` зі значенням `module`:

```
<script src="./main.js" type="module"></script>
```

Після цього викликати `mockWithVideo`, вказавши шлях до попередньо записаного відео (рис. 3.4):

```
mockWithVideo("mock-video.mp4");
```

Закоментувавши `mockWithVideo`, можна знову переключитись на вебкамеру.

Вхідний потік можна узяти не лише з відео, а й із зображення – для цього додамо до бібліотеки `camera-mock.js` ще одну допоміжну функцію:

```
export const mockWithImage = (path) => {
  navigator.mediaDevices.getUserMedia = () => {
    return new Promise((resolve, reject) => {
      const canvas = document.createElement("canvas");
      const context = canvas.getContext("2d");

      const image = new Image();
      image.onload = () => {
        canvas.width = image.width;
        canvas.height = image.height;
        context.drawImage(image, 0, 0, image.width, image.height);
      }
    })
  }
}
```

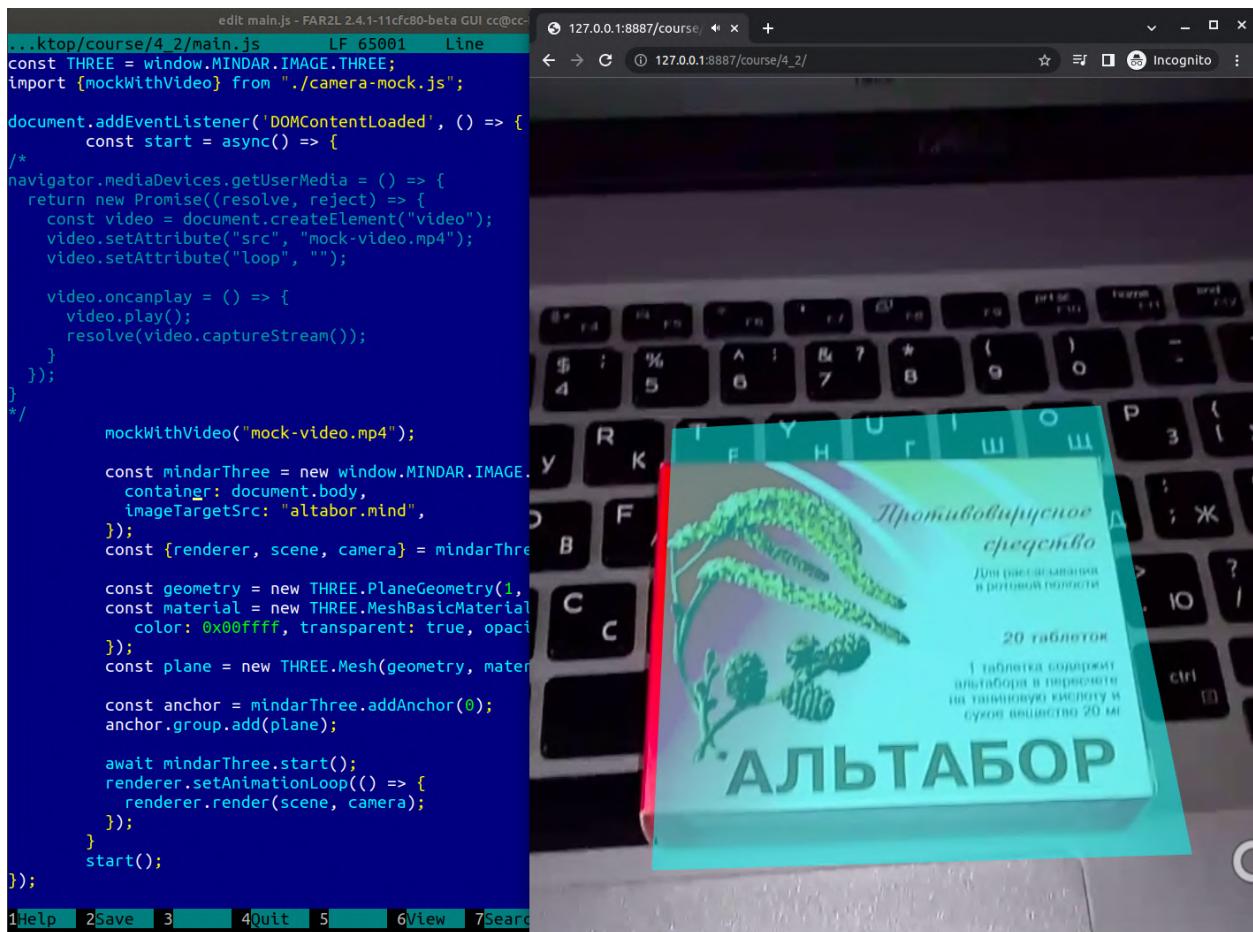


Рис. 3.4. Відстеження зображення у записаному відео.

```

        const stream = canvas.captureStream();
        resolve(stream);
    }
    image.src = path;
});
};
}

```

У функції `mockWithImage` замість об'єкту `video` створюється полотно (`canvas`), для якого метод `getContext` повертає двовимірний (2d) контекст рендеринга.

Конструктор `Image` створює новий об'єкт `HTMLImageElement` (так само, як це зробив би виклик `document.createElement('img')`), що пов'язується із відстежуваним зображенням встановленням атрибуту `src` у шлях до нього `path`.

Після успішного завантаження зображення спрацьовує метод `onload`,

за яким розміри полотна встановлюються у розміри зображення, а саме зображення викликом `drawImage` копіюється на полотно.

Нарешті, метод `captureStream` повертає `CanvasCaptureMediaStream` – захоплення зображення на полотні.

Для використання методу `mockWithImage` на початку файлу `main.js` його необхідно імпортувати з бібліотеки `camera-mock.js` командою

```
import {mockWithImage} from "./camera-mock.js";
```

Після цього викликати `mockWithImage`, вказавши шлях до файлу зображення:

```
mockWithImage("mock-image.png");
```

### 3.3 Застосування контейнеру для WebAR

Зазвичай WebAR додатки працюють у повноекранному режимі, проте не завжди це є доцільним.

Ми можемо легко перетворити елемент HTML на AR контейнер (рис. 3.5):

```
<html>
  <head>
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
      ></script>
    <script src=".main.js" type="module"></script>
    <style>
      #my-ar-container {
        height: 300px;
        width: 300px;
        position: relative;
        overflow: hidden;
      }
    </style>
```

```

</head>
<body>
    <h1>Моя WebAR сторінка</h1>
    <div id="my-ar-container"></div>
    <p>Вона працює!</p>
</body>
</html>

```

Контейнер створюється командою `div` та ідентифікується ім'ям `my-ar-container` – саме його необхідно вказати в якості значення параметру `container` конструктору `MindARThree`:

```

const mindarThree = new window.MINDAR.IMAGE.MindARThree({
    container: document.querySelector("#my-ar-container"),
    imageTargetSrc: "altabor.mind",
});

```

За замовчуванням висота контейнера дорівнює нулю. Змінити параметри контейнера `my-ar-container` можна, визначивши для нього стиль CSS. У блоці `<style></style>` він налаштований встановленням ширини (`width`) та висоти (`height`) у певні значення в пікселях, а позиціювання (`position`) – у відносне (`relative`). Через те, що відеопотік може мати інше співвідношення сторін, ніж у контейнера, частина відео з'явиться за межами області контейнера – щоб цього недопустити, властивість переповнення (`overflow`) встановлена у значення `hidden` (приховувати переповнення).

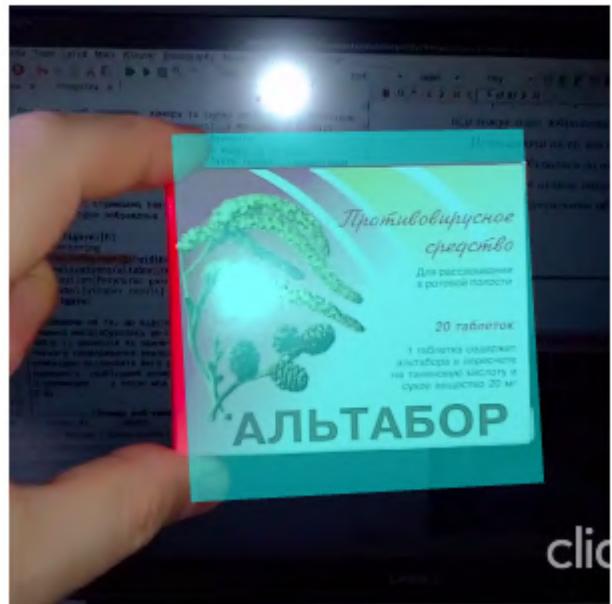
### 3.4 Завантаження текстур

Текстури, як правило, являють собою зображення, які найчастіше створюються в сторонній програмі, такій як Photoshop або GIMP. Наприклад, для розміщення зображення з локального файлу `coke.png` на площині достатньо зробити наступні мінімальні модифікації:

- 1) створити об'єкт класу `TextureLoader`:

```
var loader = new THREE.TextureLoader();
```

## Моя WebAR сторінка



Вона працює!

Рис. 3.5. Убудування доповненої реальності в елемент веб-сторінки.

2) при визначенні матеріалу площини вказати, що властивість `map` містить завантажену за допомогою методу `load` класу `TextureLoader` текстуру:

```
const material = new THREE.MeshBasicMaterial({  
    color: 0xffffffff,  
    map: loader.load("coke.png")  
});
```

Якщо текстура невелика за розміром, вона завантажиться досить швидко, проте для великих текстур буде помітно, що, доки текстура не буде завантажена, площа не з'явиться (і взагалі весь подальший код виконуватися не буде). Для того, щоб завантаження текстури не призупиняло виконання програми, використовують зворотний виклик – функцію, яка буде викликана після завершення завантаження текстури. Повертаючись до попереднього прикладу, ми можемо дочекатися завантаження текстури, перш ніж додавати об'єкт до сцени, наступним чином:

```

var plane;

const anchor = mindarThree.addAnchor(0);
const loader = new THREE.TextureLoader();
loader.load("coke.png", (texture) => {
    const geometry = new THREE.PlaneGeometry(1, 1);
    const material = new THREE.MeshBasicMaterial({
        color: 0xffffffff,
        map: texture
    });
    plane = new THREE.Mesh(geometry, material);
    anchor.group.add(plane);
});

await mindarThree.start();
renderer.setAnimationLoop(() => {
    plane.rotation.z+=0.01;
    renderer.render(scene, camera);
});

```

`plane` оголошена як змінна для того, щоб її можна було використати поза межами функції зворотного виклику, де вона стає об'єктом класу `Mesh`, наприклад, у функції анімації, в якій для кожного доступного кадру змінюється її властивість `rotation` для обертання навколо перпендикулярної до полотна вісі `z` (рис. 3.6, 3.7).

Для того, щоб використовувати зображення з інших серверів, ці сервера повинні відправляти правильні заголовки. Cross-Origin Resource Sharing (CORS) – механізм, що використовує додаткові HTTP-заголовки для того, щоб надати можливість агенту користувача отримувати дозволи на доступ до обраних ресурсів із сервера в домені, відмінному від того, що сайт використовує у даний момент. Якщо такого дозволу не отримано, зображення не буде доступне для використання у `Three.js`. Якщо ви не керуєте сервером, на якому розміщені зображення, і він не відправляє заголовки дозволів, ви не зможете використовувати зображення з цього сервера. Наприклад `imgur`, `flickr` та `github` надають відповідні дозволи, проте більшість інших

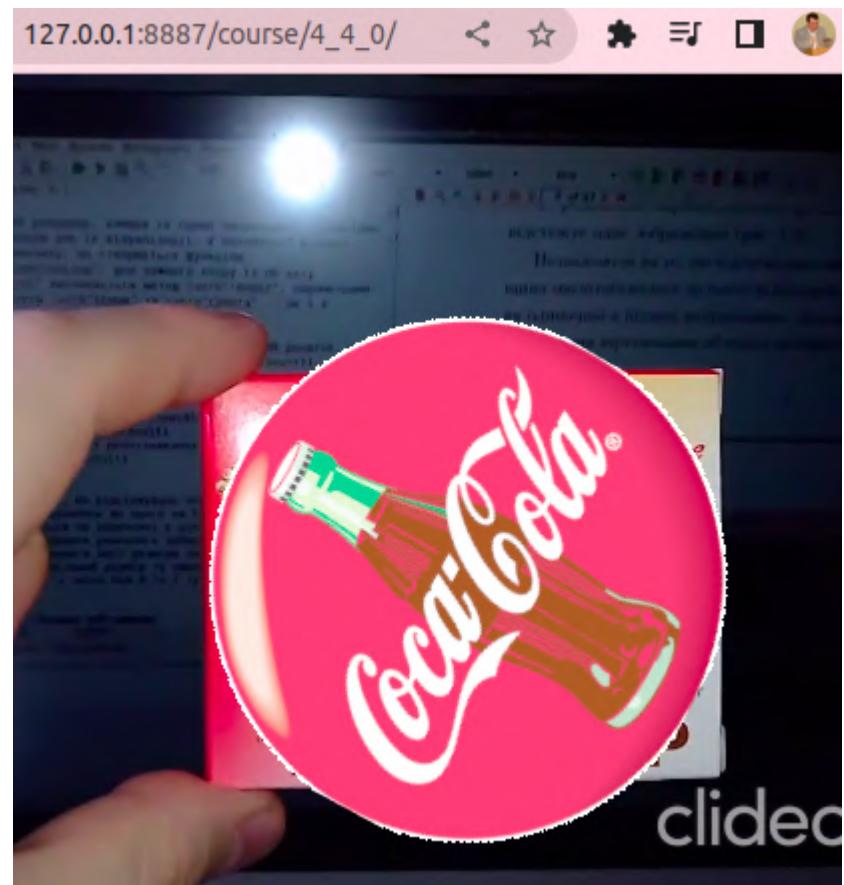


Рис. 3.6. Площина, що обертається, із накладеною текстурою.

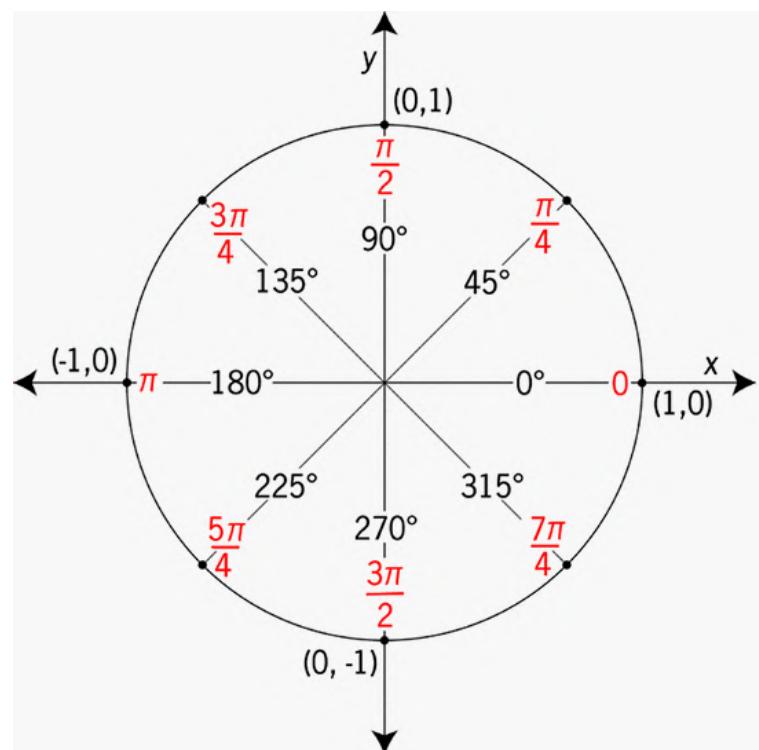


Рис. 3.7. Схема визначення параметрів обертання.

сайтів цього не роблять.

Наприклад, додамо до сцени куб, кожна грань якого міститиме власну текстуру (рис. 3.8):

```
// створюємо куб одиничного розміру
const cubegeometry = new THREE.BoxGeometry(1, 1, 1);

// масив гіперпосилань на текстири дляожної з граней
const textures = [
    "https://live.staticflickr.com/854/42936929215_efa87c8a9a_b.jpg",
    "https://live.staticflickr.com/514/18832759790_bed1aeece8_b.jpg",
    "https://live.staticflickr.com/6140/5934453114_3675350a78_b.jpg",
    "https://live.staticflickr.com/2342/2269094999_6ac65c0947.jpg",
    "https://live.staticflickr.com/1757/41821018565_614db58ddc_b.jpg",
    "https://live.staticflickr.com/3103/2420240470_bc0bb7a260.jpg"
]

// масив для завантажуваних текстур
var cubematerials = [];

for(let i=0; i<7; i++)
    if(i!=6) // дляожної з граней із номерами від 0 до 5
        // очікуємо на завантаження текстири
        loader.load(textures[i], (texture) => {
            // і додаємо її до масиву завантажуваних текстур
            cubematerials.push(new THREE.MeshBasicMaterial({map:texture}));
        });
    else { // коли усі текстири завантажені
        // об'єднуємо геометрію та матеріал
        const cubemesh = new THREE.Mesh(cubegeometry, cubematerials);
        // зміщуємо куб вправо на півтори ширини площини
        cubemesh.position.set(1.5, 0, 0);
        // додаємо куб до тієї ж групи, де розташована площа
        anchor.group.add(cubemesh);
    }
}
```

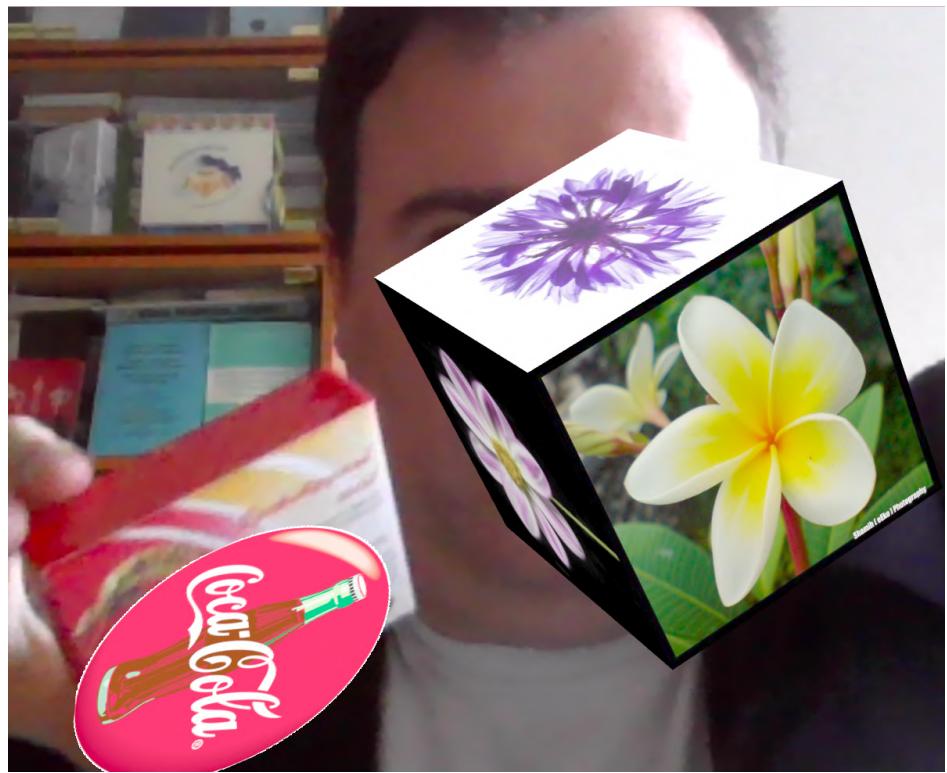


Рис. 3.8. Текстуровані площини та куб.

Однак слід зазначити, що за замовчуванням єдиною геометрією, яка підтримує декілька матеріалів, є BoxGeometry.

### 3.5 Завантаження моделей

Для того, щоб завантажити модель за допомогою Three.js, ії потрібно створити або узяти готову безкоштовну модель в одному із поширених форматів – наприклад, GLTF – із сайтів на кшалт Turbosquid.

У репозитарію Three.js наявні велика кількість завантажувачів моделей, проте не всі вони входять до складу бібліотеки – чимало їх винесені у приклади (<https://unpkg.com/browse/three/examples/jsm/loaders/>), а тому потребують окремого завантаження.

Зокрема, для завантаження моделей у форматі GLTF необхідний файл GLTFLoader.js, який знаходиться за вказаним вище шляхом. Застосування цього файлу вимагає, у свою чергу, імпортuvання бібліотеки Three.js для того (яка вже є частиною MindAR). Для того, щоб не імпортuvати одну й ту же бібліотеку двічі (причому цілком ймовірно, що у різних версіях), додамо у документ HTML таблицю імпорту імен:

```

<script type="importmap">
{
  "imports": {
    "three": "https://unpkg.com/three/build/three.module.js",
    "GLTFLoader":
      "https://unpkg.com/three/examples/jsm/loaders/GLTFLoader.js"
  }
}
</script>

```

Визначення імені `three` виконується для задоволення залежностей класу `GLTFLoader` у наступному коді:

```

const THREE = window.MINDAR.IMAGE.THREE;

import {GLTFLoader} from "GLTFLoader";

document.addEventListener("DOMContentLoaded", () => {
  const start = async() => {
    const mindarThree = new window.MINDAR.IMAGE.MindARThree({
      container: document.body,
      imageTargetSrc: "https://raw.githubusercontent.com/hiukim/"+ "mind-ar-js/master/examples/image-tracking/assets/"+"band-example/raccoon.mind",
    });
    const {renderer, scene, camera} = mindarThree;
    const anchor = mindarThree.addAnchor(0);
    const loader = new GLTFLoader();
    loader.load("https://raw.githubusercontent.com/hiukim/"+ "mind-ar-js/master/examples/image-tracking/assets/"+"band-example/raccoon/scene.gltf", (gltf) => {
      gltf.scene.scale.set(0.1, 0.1, 0.1);
    });
  };
});

```

```

        gltf.scene.position.set(0, -0.4, 0);
        anchor.group.add(gltf.scene);
    });

    await mindarThree.start();
    renderer.setAnimationLoop(() => {
        renderer.render(scene, camera);
    });
}
start();
});

```

Імпорт класу `GLTFLoader` виконується із положення, визначеного у карті імпорту за ім'ям `GLTFLoader`. Шлях, вказаний у `imageTargetSrc`, є гіперпосиланням на файл із описом опорних точок відстежуваного зображення <https://github.com/hiukim/mind-ar-js/blob/master/examples/image-tracking/assets/band-example/raccoon.png>.

Для завантаження використовується метод `load(url, onLoad, onProgress, onError)` об'єкту `loader` класу `GLTFLoader`, що має один обов'язковий та 3 необов'язкові аргументи:

- `url` – рядок, що містить шлях/URL-адресу файлу моделі (обов'язковий аргумент);
- `onLoad` – функція, що буде викликана після успішного завершення завантаження (у якості аргументу ця функція отримує завантажений `Object3D`);
- `onProgress` – функція, що буде викликатися, доки йде процес завантаження. Аргументом буде об'єкт `XMLHttpRequest`, що міститиме відомості про стан завантаження (`total`, `loaded` у байтах);
- `onError` – функція, що буде викликана при помилці завантаження.

У коді показано зворотний виклик `load` із одним параметром – шляхом до моделі. Після завершення завантаження моделі (змінна `gltf`) виконується її масштабування (`scale.set`), позиціювання (`position.set`) та додавання до якірної групи (`anchor.group.add`) об'єкту `gltf.scene`.

Згідно документації із GLTFLoader (<https://threejs.org/docs/index.html?q=Loader#examples/en/loaders/GLTFLoader>), змінна `gltf` містить 5 властивостей: `animations` – масив об'єктів класу `AnimationClip`; `scene` – груповий об'єкт (класу `Group`); `scenes` – масив групових об'єктів; `cameras` – масив камер; `asset` – текстури.

Масштабування моделі та встановлення її координат відносно відстежуваного зображення є результатом експерименту (рис. 3.9).

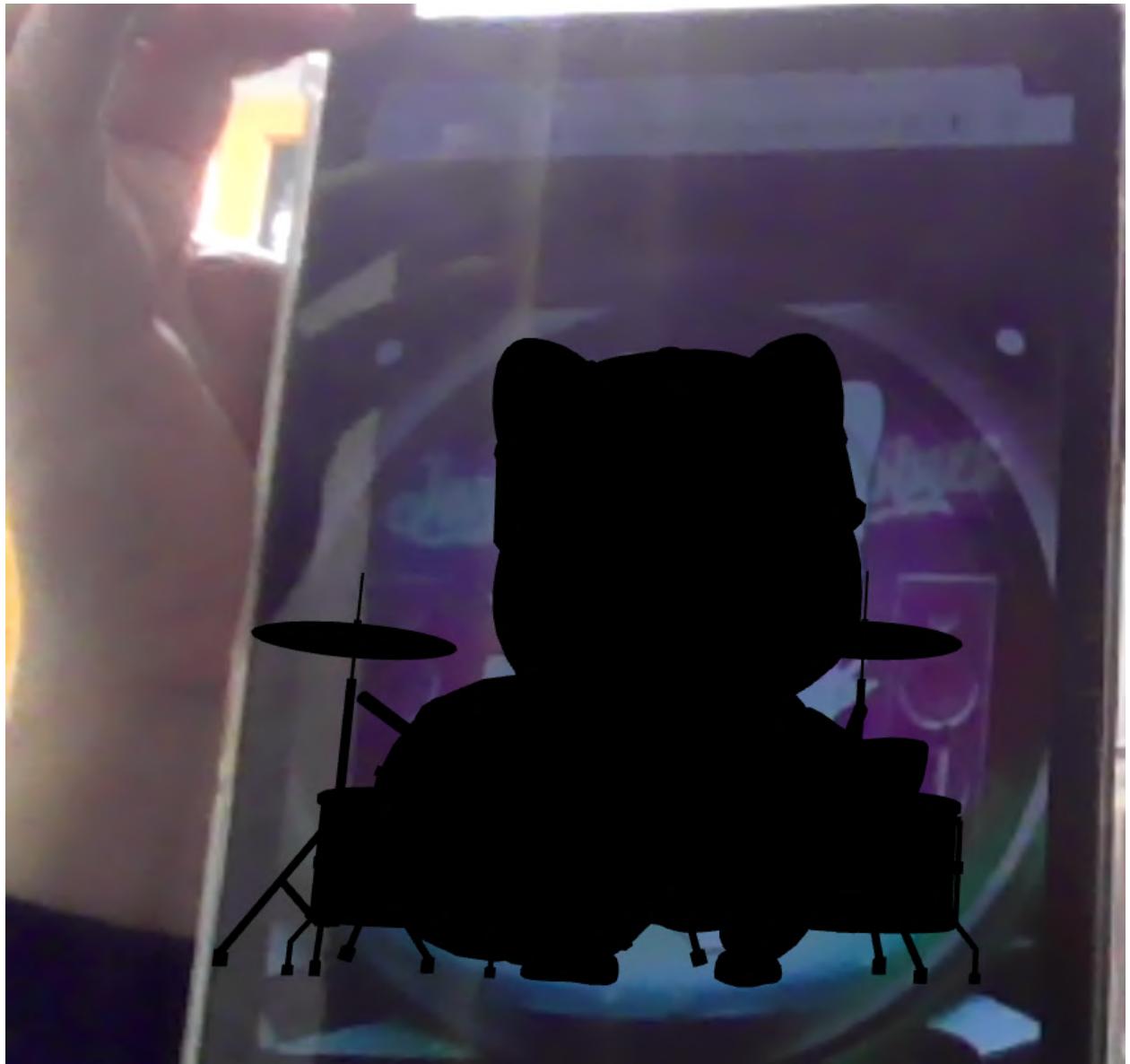


Рис. 3.9. Завантаження моделі.

Модель виглядає абсолютно темною, тому що на сцені, зображеній на полотні, ніч – відсутнє освітлення. Раніше це було непомітно, тому що у передніх прикладах використовувався тип матеріалу `MeshBasicMaterial`, колір якого не залежить від освітлення, проте це скоріше виключення, ніж

правило.

Three.js надає можливість використання таких основних типів матеріалів (рис. 3.10):

**LineBasicMaterial** – матеріал для відображення каркасних (проволочних) об'єктів;

**LineDashedMaterial** – матеріал для відображення каркасних (проволочних) об'єктів з пунктирним каркасом;

**MeshBasicMaterial** – матеріал для відображення об'єктів із простою (пласкою або каркасною) тінню без можливості освітлення;

**MeshDepthMaterial** – матеріал для відображення об'єктів за глибиною: близчі до камери частини об'єкту світліші, дальші – темніші.

**MeshLambertMaterial** – матеріал для відображення таких об'єктів, як необроблена деревина або камінь (небліскучі поверхні);

**MeshMatcapMaterial** – матеріал, який визначається текстурою у форматі MatCap (або Lit Sphere), яка кодує колір і затінення матеріалу. **MeshMatcapMaterial** не реагує на світло, оскільки файл зображення у форматі MatCap вже має власне освітлення.

**MeshNormalMaterial** – матеріал, що відображає нормальні вектори дотичних до об'єкту площин у RGB-кольорах;

**MeshPhongMaterial** – матеріал для бліскучих поверхонь із дзеркальними відблисками (лакована деревина);

**MeshStandardMaterial** – стандартний для багатьох 3D-програм, таких як Unity, Unreal 3D та 3D Studio Max фізичний матеріал;

**MeshToonMaterial** – розширення **MeshPhongMaterial** із відображенням тіней;

**PointsMaterial** – матеріал для об'єктів класу Points;

**ShaderMaterial** – матеріал для відображення користувачьким шейдером (програма, що написана на GLSL, а виконується на GPU);

**RawShaderMaterial** – працює так само, як і **ShaderMaterial**, за винятком того, що визначення будованих форм та атрибутів не додаються автоматично до коду GLSL-шейдерів;

**ShadowMaterial** – матеріал, що дає тіні, але в усьому іншому є повністю прозорим;

**SpriteMaterial** – матеріал для об'єктів класу Sprite.

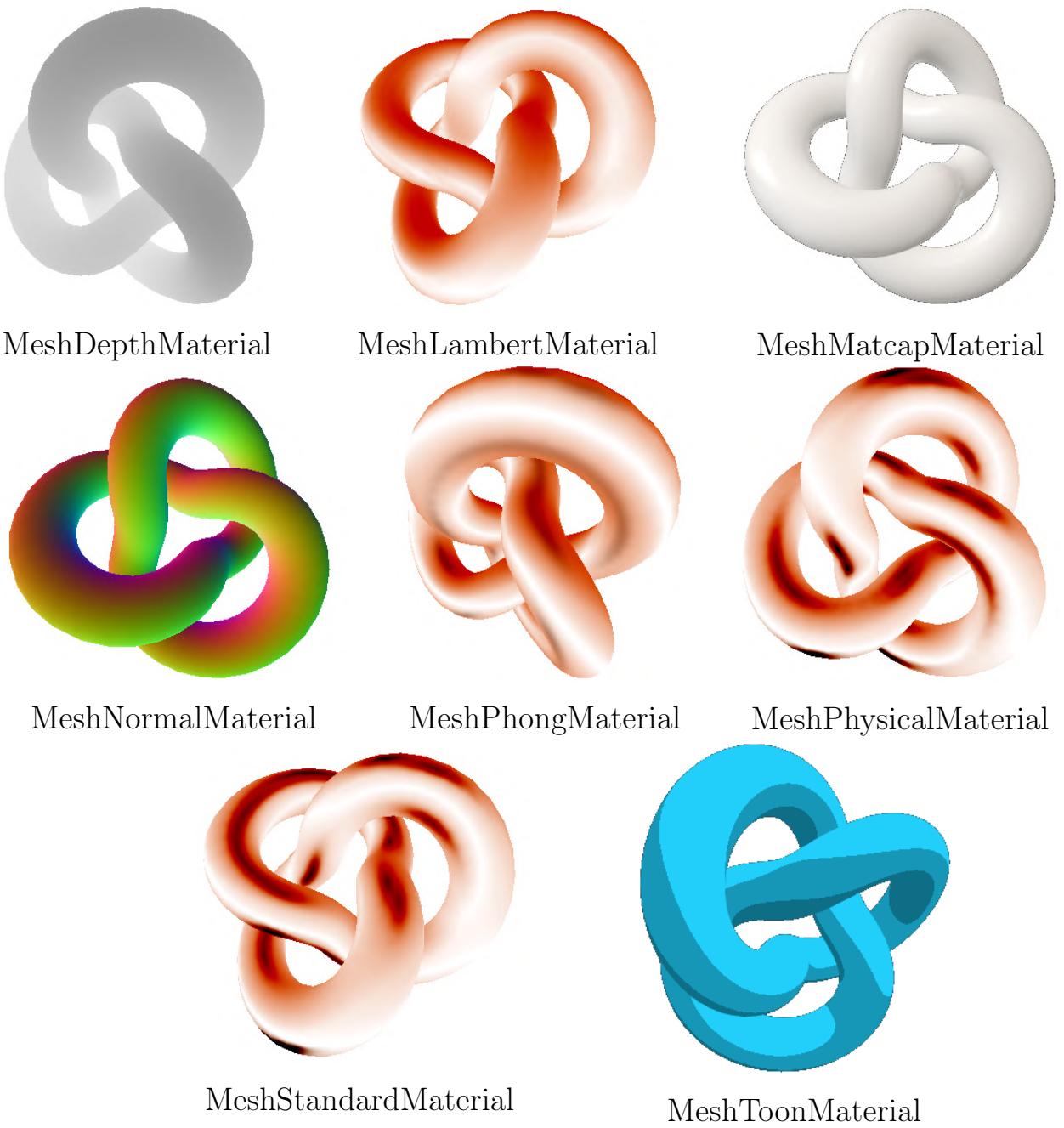


Рис. 3.10. Основні типи матеріалів.

Наша модель складається з матеріалів, які відбиваючи світло, тому необхідно додати його джерело. Three.js пропонує такі основні джерела світла (рис. 3.11):

**AmbientLight** – всі об'єкти сцени освітлюються рівномірно. Конструктор класу **AmbientLight** приймає два параметри – колір та інтенсивність. Так, для того, щоб сцена була краще освітлена, достатньо змінити другий параметр. Мінімальна інтенсивність (0) відповідатиме повністю затемненій сцені, на якій буде видимою лише площа через відповідний матеріал, максимальна (1) – повністю освітленій сцені.

**DirectionalLight** – всі промені йдуть паралельно у вказано напрямі (денне світло). Для такого типу освітлення напрям освітлення визначається цільовим об'єктом, що буде освітлюватись.

**HemisphereLight** – джерело світла знаходиться над сценою, при затіненні кольори змінюються від кольору неба до кольору землі. Конструктор класу напівсферичного освітлення приймає три параметри – колір неба, колір землі та інтенсивність освітлення: напівсферичне освітлення знаходиться над сценою, а колір освітлення змінюється від кольору неба до кольору землі.

**PointLight** – точкове джерело, промені якого поширюються в усіх напрямах (лампочка). Конструктор класу **PointLight** має чотири параметри: колір (за замовчанням білий – 0xffffffff), інтенсивність (за замовчанням максимальну – 1), дальність світла (за замовчанням 0 – без обмежень), коефіцієнт затухання світла зі збільшенням відстані (за замовчанням 1). Точкове освітлення залежить від того, де знаходиться джерело освітлення.

**RectAreaLight** – світло поширюється через прямокутник (яскраве вікно, відкриті освітлені двері).

**SpotLight** – світло поширюється з точки в одному напрямі вздовж конуса (прожектор). Так само, як і для напрямленого освітлення, прожекторне потребує вказання освітлюваного об'єкту.

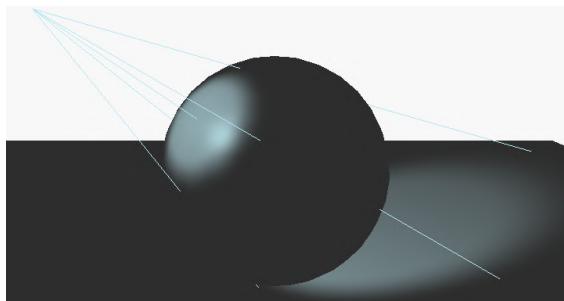
Додавши до сцени напівсферичне освітлення, побачимо суттєво більше деталей (рис. 3.12):

```
var light = new THREE.HemisphereLight(0xffffff, 0xbbbbff, 1);
scene.add(light);
```

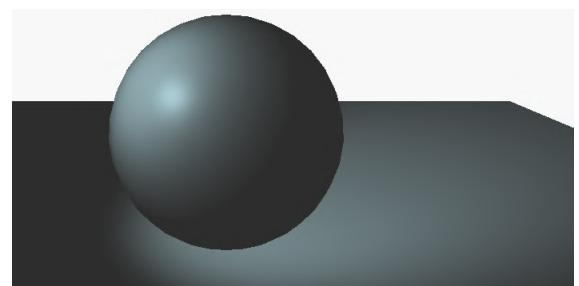
Код для завантаження моделей використовується досить часто, тому можна оформити його у вигляді допоміжної функції `loadGLTF` та винести у нову бібліотеку – `loader.js`:

```
import {GLTFLoader} from
"https://unpkg.com/three/examples/jsm/loaders/GLTFLoader.js";
import * as THREE from
"https://unpkg.com/three/build/three.module.js";

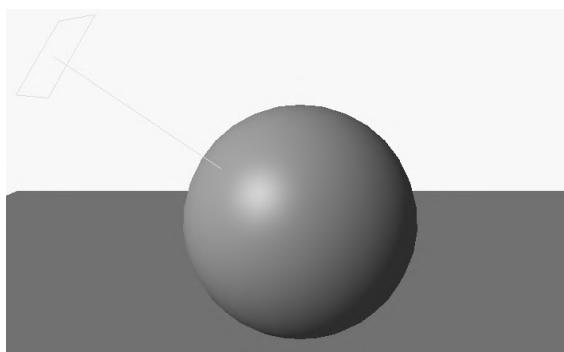
export const loadGLTF = (path) => {
```



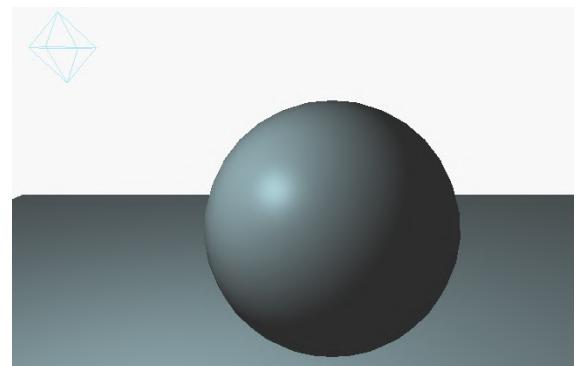
SpotLight



AmbientLight



DirectionalLight



PointLight

Рис. 3.11. Деякі типи освітлення.

```
return new Promise((resolve, reject) => {
  const loader = new GLTFLoader();
  loader.load(path, (gltf) => {
    resolve(gltf);
  });
});
```

}

Тоді у карті імпорту залишиться лише одне ім'я:

```
<script type="importmap">
{
  "imports": {
    "three": "https://unpkg.com/three/build/three.module.js"
  }
}
</script>
```

У файлі main.js команда імпорту зміниться на



Рис. 3.12. Освітлена модель.

```
import {loadGLTF} from "./loader.js";
```

А для завантаження моделі використовується вже створена допоміжна функція:

```
const gltf = await loadGLTF("https://raw.githubusercontent.com/"+  
    "hiukim/mind-ar-js/master/examples/image-tracking/assets/"+  
    "band-example/raccoon/scene.gltf");  
gltf.scene.scale.set(0.1, 0.1, 0.1);  
gltf.scene.position.set(0, -0.4, 0);  
anchor.group.add(gltf.scene);
```

### 3.6 Відстеження декількох зображень

Код із попереднього прикладу надавав можливість завантажити модель, відстежуючи зображення <https://raw.githubusercontent.com/hiukim/mind-ar-js/master/examples/image-tracking/assets/band-example/raccoon.png>:

```
const THREE = window.MINDAR.IMAGE.THREE;

import {loadGLTF} from "./loader.js";

document.addEventListener('DOMContentLoaded', () => {
  const start = async() => {
    const mindarThree = new window.MINDAR.IMAGE.MindARThree({
      container: document.body,
      imageTargetSrc: "https://raw.githubusercontent.com/hiukim/" +
        "mind-ar-js/master/examples/image-tracking/assets/" +
        "band-example/raccoon.mind",
    });
    const {renderer, scene, camera} = mindarThree;

    var light = new THREE.HemisphereLight(0xfffffff, 0xbfffff, 1);
    scene.add(light);

    const gltf = await loadGLTF("https://raw.githubusercontent.com" +
      "/hiukim/mind-ar-js/master/examples/image-tracking/assets/" +
      "band-example/raccoon/scene.gltf");
    gltf.scene.scale.set(0.1, 0.1, 0.1);
    gltf.scene.position.set(0, -0.4, 0);

    const anchor = mindarThree.addAnchor(0);
    anchor.group.add(gltf.scene);

    await mindarThree.start();
    renderer.setAnimationLoop(() => {
```

```

    renderer.render(scene, camera);
  });
}

start();
});

```

Додамо можливість відстеження ще одного зображення – <https://raw.githubusercontent.com/hiukim/mind-ar-js/master/examples/image-tracking/assets/band-example/bear.png>, за знаходженням якого завантажимо іншу модель – <https://raw.githubusercontent.com/hiukim/mind-ar-js/master/examples/image-tracking/assets/band-example/bear/scene.gltf>.

Для цього необхідно додати відповідний код після завантаження першої моделі:

```

const newmodel = await loadGLTF("https://raw.githubusercontent.com"+
  "/hiukim/mind-ar-js/master/examples/image-tracking/assets/" +
  "band-example/bear/scene.gltf");
newmodel.scene.scale.set(0.1, 0.1, 0.1);
newmodel.scene.position.set(0, -0.4, 0);

const newanchor = mindarThree.addAnchor(1);
newanchor.group.add(newmodel.scene);

```

Параметром методу `addAnchor` для другої моделі є не 0, а 1 – це буде номер другого зображення. Для того, щоб їх розрізнати, яке з них перше, а яке – друге, необхідно їх завантажити до компілятора зображень у правильному порядку: першим – зображення єнота, а другим – ведмедя (рис. 3.13).

Результат роботи компілятора – один файл `targets.mind`, що міститиме опорні точки для двох зображень у визначеному порядку: саме його ім'я необхідно передати другим параметром (`imageTargetSrc`) конструктора `MindARThree`:

```

const mindarThree = new window.MINDAR.IMAGE.MindARThree({
  container: document.body, imageTargetSrc: "targets.mind",
});

```

# Image Targets Compiler

Select target images and start

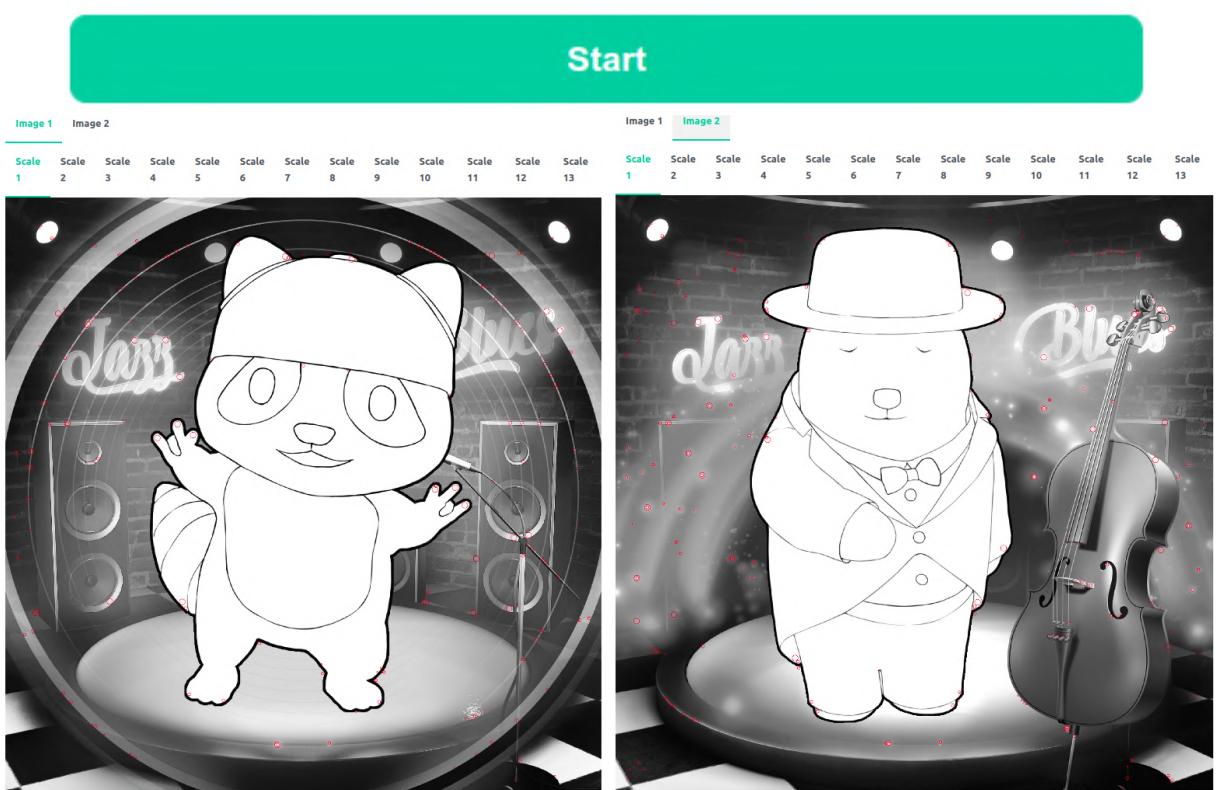
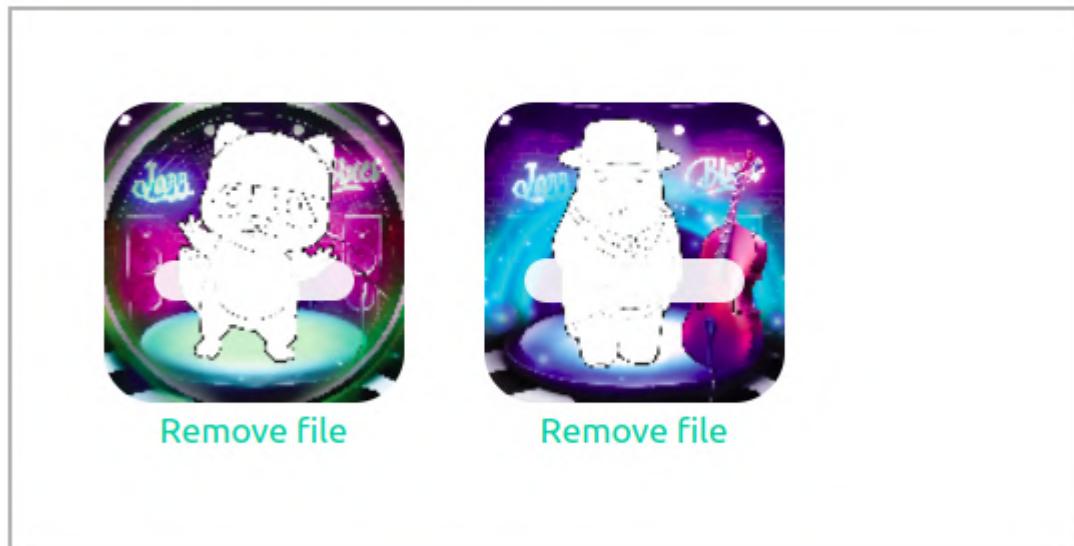


Рис. 3.13. Генерування файлу з опорними точками за декількома зображеннями.

Тестування показує, що кожне із зображенень успішно розпізнається, якщо їх демонструвати окремо, проте не одночасно: якщо перше зо-

зображення розпізнано, друге не виявляється; якщо перше зображення не розпізнано, друге виявляється (рис. 3.14).



Рис. 3.14. Розпізнавання одного із декількох зображень.

Процес відстеження об'єктів складається з двох фаз (табл. 3.1).

Перша фаза – це *виявлення* (detection) об'єктів, що відбувається, на-

Таблиця 3.1

Порівняльна характеристика виявлення та відстеження.

Характеристика	Виявлення	Відстеження
обчислювальна ємність	висока	низька
швидкість	висока	низька
чутливість до точності	висока	низька
масштабованість до багатьох зображень	висока	низька
обмеження MindAR на кількість цілей	30	3
обмеження 8th Wall на кількість цілей	1000	10

приклад, при першому запуску програму, коли цільове зображення ще не знайдено. У процесі виявлення бібліотека визначає опорні точки у ведеопотоку з камери і намагатися знайти збіг в базі даних зображень.

Як тільки ціль буде знайдена, бібліотека переходить до фази *відстеження* (tracking) замість пошуку, базуючись на раніше виявленому місці розташування об'єкту. Під час відстеження бібліотека намагається слідувати за ціллю від кадру до кадру.

Виявлення, як правило, більш обчислювально ресурсоємне, тому що воно передбачає багато роботи із пошуку і зіставлення.

Хоча виявлення займає більше часу для обчислень, деякі кадри можуть бути пропущені, тому що у процесі виявлення для користувача немає суттєвої затримки, займе початкове виявлення об'єкту чверть секунди чи півсекунди.

При відстеженні необхідно підтримувати високу частоту кадрів та мінімізувати пропуски для точного суміщення відстежуваного реального та віртуального об'єктів. У цьому сенсі виявлення є менш чутливою до швидкості опрацювання, ніж відстеження.

Виявлення чутливе до точності задля недопущення помилкових спрощувань. При відстеженні ж втрата одного кадру не суттєво вплине, тому що ми можна виконати апроксимацію, взявши ковзне середнє з декількох кадрів.

Алгоритм виявлення набагато більш масштабований до декількох цілей, ніж алгоритм відстеження – фактично, він лінійно залежний від їх кількості, у той час як при відстеженні додавання кожної нової цілі по-

двоює обчислювальні витрати.

Вказані у табл. 3.1 обмеження на кількість детектованих та відстежуваних цілей є орієнтованим та суттєво залежать від обладнання та середовища, в яких застосовуються WebAR додатки, створені у MindAR та 8th Wall (<https://www.8thwall.com/>) – одному з лідерів на ринку комерційних WebAR SDK.

Підвищення кількості одночасно відстежуваних зображень без суттєвих обчислювальних витрат можливе за умови спрощення та структурування цільового зображення – саме так досягається баланс швидкості роботи та кількості відстежуваних цілей у рушіях на основі маркерів JSARToolKit.

Для того, щоб одночасно виявляти більш ніж одну ціль у MindAR, необхідно встановити третій параметр конструктора `MindARThree` – змінну `maxTrack` (максимальну кількість цілей, яка має відстежуватись у будь-який момент часу) – у необхідне значення (рис. 3.15):

```
const mindarThree = new window.MINDAR.IMAGE.MindARThree({
    container: document.body, imageTargetSrc: "targets.mind",
    maxTrack: 2,
});
```



Рис. 3.15. Одночасне відстеження декількох зображень.

### 3.7 Анімація моделей

Створюючи моделі, 3D-митці часто закладають у них можливості анімації – так, в моделях у форматі GLTF за них відповідає властивість `animations`, що являє собою масив об'єктів класу `AnimationClip`. Дізнатись, чи є модель анімована, можна за довжиною цього масиву (властивість `animations.length`) – якщо він порожній, модель статична.

У документації до Three.js (<https://threejs.org/docs/#manual/en/introduction/Animation-system>) описано нову систему анімації, подібну до використовуваній у нових версіях Unity та Unreal Engine. Її основою є анімаційний програвач `AnimationMixer`, із яким пов'яжемо завантажену модель:

```
const mixer = new THREE.AnimationMixer(gltf.scene);

if(gltf.animations.length != 0)  {
    const action = mixer.clipAction(gltf.animations[0]);
    action.play();
}
```

Створення об'єкту `mixer` у такий спосіб є безпечним незалежно від того, є модель анімованою чи статичною. Метод `clipAction` приймає в якості параметру першу з анімацій та повертає об'єкту `action` класу `AnimationAction`, який й активується викликом `play`.

Виклик `play` не обов'язково означає, що анімація почнеться негайно – вона може бути відкладеною. Більш того, анімація може завершитись до того, як модель буде показано.

Інша проблема – нерівномірність часу, що виділяється на роботу програми веб-браузером та виражається у несталій швидкості відеопотоку (FPS – кадрів на секунду).

Для їх подолання скористаємося таймером Three.js та модифікуємо цикл анімації (рис. 3.16):

```
const clock = new THREE.Clock();

await mindarThree.start();
```

```
renderer.setAnimationLoop(() => {
    const delta = clock.getDelta();
    mixer.update(delta);
    renderer.render(scene, camera);
});
```



Рис. 3.16. Анімована модель.

Метод `getDelta` класу `Clock` повертає час, що пройшов зі створення таймеру або попереднього звернення до нього – саме цей час й передається методу `update` анімаційного міксеру для того, щоб відтворити відповідний фрагмент анімації.

Інший спосіб анімації 3D-вмісту є зовнішнім по відношенню до моделі та передбачає зміну таких властивостей, як координати (`position`), кут (`rotation`), масштаб (`scale`), прозорість (`opacity`) та ін.

Наприклад, для обертання моделі навколо вісі у можна вставити у цикл анімації наступний код:

```
gltf.scene.rotation.set(0, gltf.scene.rotation.y+delta, 0);
```

Тут метод `set` властивості `rotation` застосовується для зміни кута повороту відносно другої координати (`y`), заданої у радіанах, на значення часу, що пройшов з попереднього виклику `getDelta`. Простіше можна записати так:

```
gltf.scene.rotation.y += delta;
```

У такий спосіб можна створити багато цікавих ефектів, навіть якщо модель є статичною.

### 3.8 Опрацювання подій

Досить часто буває необхідно зафіксувати момент виявлення цільового зображення або його втрати камерою. Так, можна запускати анімацію моделі не одразу, а лише тоді, коли модель з'являється, та зупиняти її з поверненням до початковим налаштувань, коли вона щезає.

Традиційний спосіб це зробити у JavaScript – налаштовувати функцій для опрацювання відповідних подій за допомогою `addEventListener`. Бібліотека MindAR містить ряд допоміжних функцій для цього. Зокрема, у якорі є можливість перевизначити дві функції – `onTargetFound` та `onTargetLost`:

```
anchor.onTargetFound = () => {
    console.log("ціль виявлена");
}

anchor.onTargetLost = () => {
    console.log("ціль втрачена");
}
```

При виявленні декількох зображень для кожного з них можна створити свій якор та визначити відповідні функції для опрацювання подій виявлення та втрати цільового зображення – це дуже зручно як для налагодження

програми, так й для доцільного управління. При роботі з озвученим відео без його призупинення при втраті цілі не обйтись – якщо це не зробити, воно буде продовжувати програватись, і, хоча ми його вже не бачимо, аудіо продовжуватиме звучати.

Продемонструємо це на прикладі додавання аудіо до моделі. Визначимо та розмістимо у бібліотеці створимо `loader.js` ще одну допоміжну функцію:

```
export const loadAudio = (path) => {
  return new Promise((resolve, reject) => {
    const loader = new THREE.AudioLoader();
    loader.load(path, (buffer) => {
      resolve(buffer);
    });
  });
}
```

Функція `loadAudio` дуже схожа на визначену раніше `loadGLTF`, за виключенням того, що клас `AudioLoader` є частиною бібліотеки `Three.js`, а не стороннім модулем.

На початку `main.js` експортуємо обидві функції:

```
import {loadGLTF, loadAudio} from "./loader.js";
```

Після завантаження моделі завантажимо аудіо-файл - локальний або з Інтернет:

```
const audioClip = await loadAudio("https://archive.org/download/"+
  "Jazz_Sampler-9619/Kevin_MacLeod_-_AcidJazz.mp3");
```

Результатом завантаження є об'єкт `audioClip` класу `AudioBuffer` (частини Web Audio API, а не `Three.js`, тобто весь аудіо-файл завантажується у пам'ять пристрою, тобто потокове аудіо так не завантажити.

`AudioListener` представляє собою віртуального слухача всіх по-зиційних і непозиційних аудіоэффектів на сцені:

```
const listener = new THREE.AudioListener();
```

`listener` – об'єкт класу `AudioListener` – є обов'язковим параметром конструктора для таких аудіо-сущностей, як `Audio` та `PositionalAudio`. `AudioListener` представляє позицію та орієнтацію унікальної особи, що слухає аудіо сцену, та використовується в аудіо просторовій візуалізації (рис. 3.17).

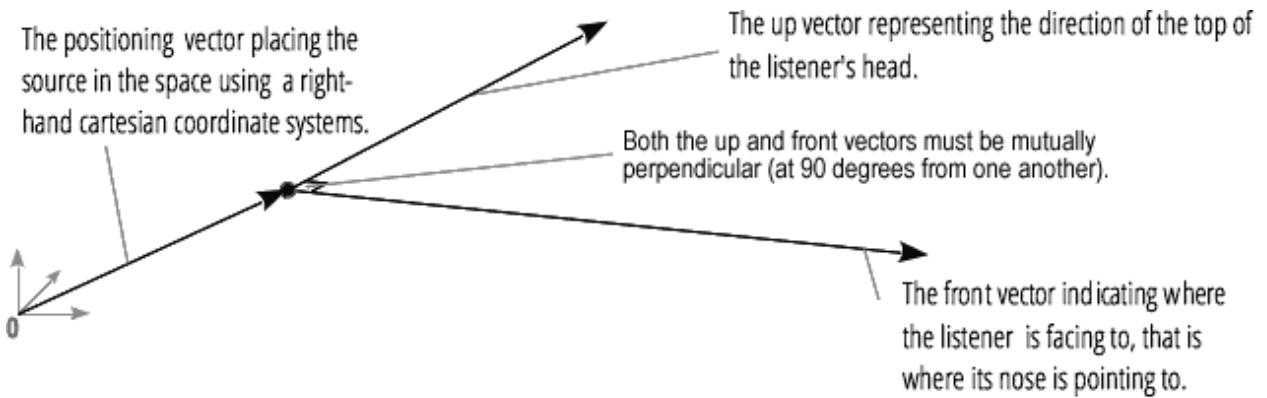


Рис. 3.17. Схема роботи просторового аудіо (за <https://developer.mozilla.org/en-US/docs/Web/API/AudioListener>).

У більшості випадків об'єкт-слухач є дочірнім об'єктом камери. Таким чином, слухач буде знаходитись у тій самій позиції, що й камера:

```
camera.add(listener);
```

Клас `Audio` відповідає за глобальне (не позиційне) аудіо, яке буде звучати завжди однаково. `PositionalAudio` використовується для створення просторового ефекту, за якого джерело звуку прив'язується до певного об'єкту – якоря цільового зображення, до якого прив'язана й модель (це створюватиме ефект звучання аудіо з місця розташування моделі):

```
const audio = new THREE.PositionalAudio(listener);
anchor.group.add(audio);
```

Встановимо, що програватиметься завантажений раніше файл:

```
audio.setBuffer(audioClip);
```

Метод `setRefDistance` встановлює еталонну відстань для зменшення гучності у міру віддалення джерела звуку від слухача, тобто відстань, на якій починає діяти зменшення гучності. Збільшення цього значення також затримує спадання гучності до тих пір, поки звук не пройде повз опорну точку:

```
audio.setRefDistance(100);
```

Передавання `true` в якості параметра методу `setLoop` закільцює відтворення аудіо:

```
audio.setLoop(true);
```

Нарешті, відтворення аудіо розпочнеться при виявленні цілі та призупиниться при її виходу з поля зору камери:

```
anchor.onTargetFound = () => {
    audio.play();
}
```

```
anchor.onTargetLost = () => {
    audio.pause();
}
```

Тепер, коли ціль виявлено, ми можемо почути фонову музику. Музика буде ставати голоснішою при наближенні цілі до камери. Якщо вона здається надто тихою, можна пропорційно збільшити її гучність:

```
listener.setMasterVolume( 20.0 * listener.getMasterVolume() );
```

### 3.9 Взаємодія з користувачем

При роботі на мобільних пристроях під управлінням iOS описаний спосіб відтворення звука (так само, як і відео), може не працювати через заборону веб-браузера Safari на автоматичне відтворення відео та аудіо без втручання користувача, що виконується функцією `start`.

Обхід цієї заборони полягає у заміні автоматичного виклику функції `start` після завантаження документа на виклик по сигналу від користувача. Наприклад, можна створити кнопку із надписом “Будь-ласка, дозвольте скористатись камерою”, по натисканню на яку викликається функція `start`:

```
//start();
const startButton = document.createElement("button");
startButton.textContent = "Будь-ласка, дозвольте"+
    " скористатись камерою";
startButton.addEventListener("click", start);
document.body.appendChild(startButton);
```

Взагалі, взаємодія з користувачем – важлива частина WebAR програм: так, типовими є опрацювання користувачем події натискання мишею на об'єкті або дотик до об'єкта на екрані мобільного телефону – `click`.

На жаль, об'єкти Three.js не є складовими DOM – об'єктної моделі документа, такими як полотно (`canvas`), – до яких можна було б отримати доступ за допомогою JavaScript, а є лише елементами полотна.

Через це доводиться опрацьовувати подію `click` на полотні та обчислювати позиції об'єктів на ньому самостійно:

```
document.body.addEventListener("click", (e) => {
// тут буде опрацьовуватись подія
});
```

Ураховуючи, що розмір контейнера, що містить полотно (у нашому випадку це тіло документа HTML – `body`), може відрізнятись навіть між запусками програми через різні розміри вікна веб-браузера, у документації зі Three.js (<https://threejs.org/docs/index.html?q=RayCaster#api/en/core/Raycaster>) рекомендується замість абсолютних координат застосовувати відносні, нормалізовані в діапазоні  $[-1; +1]$  – тоді центр полотна буде відповідати початку координат:

```
const mouseX = ( e.clientX / window.innerWidth ) * 2 - 1;
const mouseY = - ( ( e.clientY / window.innerHeight ) * 2 - 1 );
```

`clientX` та `clientY` є властивостями об'єкту `e` класу `Event`, що змінюються в діапазоні від 0 до ширини (`window.innerWidth`) та висоти вікна (`window.innerHeight`) відповідно. Ділення координати за певною віссю на довжину полотна за нею ж дає значення від 0 до 1, множення на 2 розтягує діапазон до [0; 2], а віднімання 1 зміщує до необхідного.

У вікні HTML та на полотні напрям вісі у відрізняється: у вікні вона спрямована згори донизу, а на полотні – знизу догори, тому знак “-” у другому рядку використовується для перегортання вісі `y`.

`mouseX` та `mouseY` – відносні координатами миші на полотні – утворюють двовимірний вектор:

```
const mouse = new THREE.Vector2(mouseX, mouseY);
```

Головне питання в тому, як, використовувати ці координати, визначити, чи користувач торкнувся до певного об'єкта.

Один із простих та поширеніших методів для цього – рейткастинг (ray casting – кидання променів), ідея якого полягає в тому, що ми намагаємося провести лінію (промінь) від камери до точки на екрані і далі та перевіряємо, чи не перетинається цей промінь з будь-якими об'єктами на його шляху (рис. 3.18).

Клас `Raycaster` був розроблений для реалізації методу кидання променів. Його метод `setFromCamera` встановлює положення камери, з якої виходить промінь (`camera`), та точки `mouse`, в яку він спрямовується (координати натискання кнопки миші):

```
const raycaster = new THREE.Raycaster();
raycaster.setFromCamera(mouse, camera);
```

Метод `intersectObjects` перевіряє всі перетини променя з об'єктом (якщо його другий параметр встановлений у `true`, то рекурсивно):

```
const intersects=raycaster.intersectObjects(scene.children,true);
```

Першим параметром `intersectObjects` можна було б вказати конкретний об'єкт – завантажену модель, проте вказання `scene.children` є

## Ray casting: example scenario

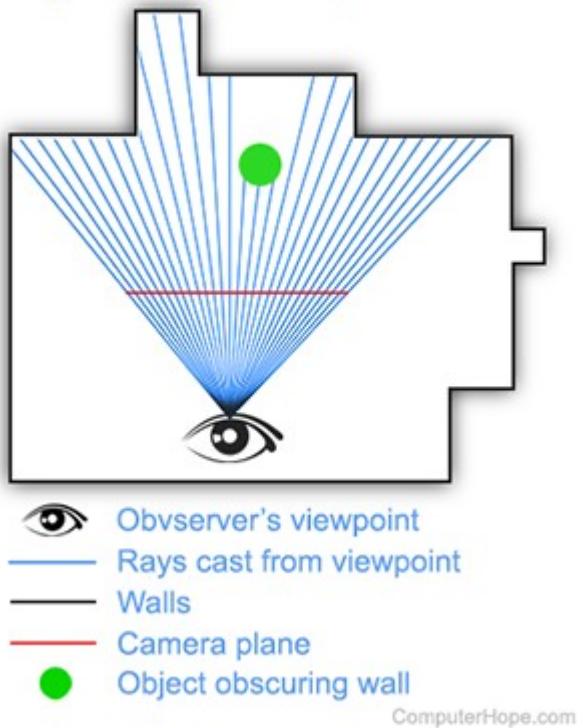


Рис. 3.18. Схема роботи методу кидання променів (за <https://www.computerhope.com/jargon/r/ray-casting.htm>).

більш універсальним, тому що надає можливість легко розширити функціональність програми для виявлення того, чи не було натиснуто кнопку миші на іншому об'єкті.

Рекурсивна перевірка необхідна через те, що сцена у Three.js являє собою ієархію об'єктів, часто вкладених один у одного: так, модель, що є складовою якоря, саме є множиною об'єктів.

Перетини із променем (спісок об'єктів) повертаються відсортовані за відстанню від камери, починаючи з найближчих. Масив перетинів має вигляд

```
[ { distance, point, face, faceIndex, object }, ... ]
```

де `distance` – відстань між початком променя та перетином; `point` – точка перетину у світових координатах; `face` – грань, з якою перетинається промінь; `faceIndex` – індекс грані, з якою перетинається промінь; `object` – об'єкт, що перетинається.

Тому, якщо спісок перетинів не порожній, нас у ньому цікавитиме перший об'єкт `intersects[0].object` – як найближчий. Через те, що це може виявитись один із об'єктів-складових моделі (яка являє собою ієархію

об'єктів Three.js), доцільним є спробувати дійти до кореневого об'єкту – самої моделі. Особливістю кореневого об'єкту є те, що для нього відсутній батьківський, тобто властивість об'єкту `parent` не містить посилання (є порожнім).

Коли на сцені декілька об'єктів накладаються, наприклад, через анімацію, доцільно увести додатковий спосіб їх розрізняння, використовуючи, наприклад, властивість `userData`, що наявна у будь-якому об'єкті Three.js – вона може бути використаний для зберігання користувачьких даних (лише даних – не функцій):

```
gltf.scene.userData.clickable = true;
```

Тут `clickable` – нове поле властивості `userData`, що й відрізнятиме шуканий об'єкт.

Виходячи з вищевикладеного, можна запропонувати такий спосіб виявлення, чи було натискання кнопки миші на моделі, для якої встановлено значення `userData.clickable`:

```
// перевірити, чи промінь, що з'єднує позицію миші та камеру,
// перетнув якісь з об'єктів
if(intersects.length>0) {
    // обрати найближчий до позиції миші об'єкт
    let o = intersects[0].object;

    // доки цей об'єкт не є кореневий та не той, що відслідковується
    while(o.parent && !o.userData.clickable)
        // підіймаємось по ієархії об'єктів Three.js вгору
        o = o.parent;

    // якщо це відслідковуваний об'єкт та є й той, що треба
    if(o.userData.clickable && o === gltf.scene)
        audio.play(); // виконуємо певну дію по даній події
}
```

Продемонструємо роботу алгоритму на прикладі відтворення звичайного (не позиційного) аудіо:

```

const THREE = window.MINDAR.IMAGE.THREE;

import {loadGLTF, loadAudio} from "./loader.js";

import {mockWithImage} from "./camera-mock.js";

document.addEventListener('DOMContentLoaded', () => {
  const start = async() => {
    const mindarThree = new window.MINDAR.IMAGE.MindARThree({
      container: document.body,
      imageTargetSrc: "https://raw.githubusercontent.com/hiukim/"+ "mind-ar-js/master/examples/image-tracking/assets/"+"band-example/raccoon.mind",
    });
  };
}

const {renderer, scene, camera} = mindarThree;

var light = new THREE.HemisphereLight(0xfffffff, 0xbfffff, 1);
scene.add(light);

const gltf = await loadGLTF("https://raw.githubusercontent.com/hiukim/mind-ar-js/master/examples/image-tracking/assets/band-example/raccoon/scene.gltf");
gltf.scene.scale.set(0.1, 0.1, 0.1);
gltf.scene.position.set(0, -0.4, 0);

gltf.scene.userData.clickable = true;

const listener = new THREE.AudioListener();
camera.add(listener);

const audioClip = await loadAudio('https://archive.org/'+ "download/FastCarSoundEffects/car%2Brace.mp3");
const audio = new THREE.Audio(listener);

```

```

audio.setBuffer(audioClip);

const anchor = mindarThree.addAnchor(0);
anchor.group.add(gltf.scene);

document.body.addEventListener("click", (e) => {
    const mouseX = ( e.clientX / window.innerWidth ) * 2 - 1;
    const mouseY = - ( e.clientY / window.innerHeight ) * 2 + 1;
    const mouse = new THREE.Vector2(mouseX, mouseY);

    const raycaster = new THREE.Raycaster();
    raycaster.setFromCamera(mouse, camera);
    const intersects =
        raycaster.intersectObjects(scene.children, true);

    if(intersects.length>0)
    {
        let o = intersects[0].object;

        while(o.parent && !o.userData.clickable)
            o = o.parent;

        if(o.userData.clickable && o === gltf.scene)
            audio.play();
    }
});

await mindarThree.start();
renderer.setAnimationLoop(() => {
    renderer.render(scene, camera);
});
}

start();
});

```

### 3.10 Відтворення відео

Як і аудіо, відео також є дуже корисним типом контенту, що часто використовується у WebAR проектах. На відміну від моделей Three.js, відео є DOM-елементом, який необхідно створити до завантаження відео:

```
export const loadVideo = (path) => {
  return new Promise((resolve, reject) => {
    const video = document.createElement("video");
    video.addEventListener("loadeddata", () => {
      video.setAttribute("playsinline", "");
      resolve(video);
    });
    video.src = path;
  });
}
```

Саме на DOM-елемент `video` покладається завантаження – за це відповідає його властивість `src`. По завершенню завантаження відео спрацює обробник події `loadeddata`, по якому для відео встановиться атрибут `playsinline` – його дія непомітна на звичайних браузерах, проте має ефект на мобільних: останні відтворюватимуть відео прямо там, де воно є, замість того, щоб за замовчуванням відкривати його в повноекранному режимі під час відтворення.

Допоміжну функцію `loadVideo` додамо до користувальської бібліотеки `loader.js` – вона може знадобитись й надалі:

```
import {loadGLTF, loadVideo} from "./loader.js";
```

В якості експерименту спробуємо згенерувати файл опорних точок за зображенням, що містить обличчя (рис. 3.19).

Відео, що буде прив'язано до цього зображення, може бути завантажено як за віддаленим URL, так і за локальним:

```
const video = await loadVideo("Sokka_on_Fortune.mkv");
```

Для управління відео необхідно створити відеотекстуру:



Рис. 3.19. Зображення з обличчям (ліворуч) та його опорні точки (праворуч) (<https://en.wikipedia.org/wiki/Sokka#/media/File:Sokka.png>).

```
const texture = new THREE.VideoTexture(video);
```

Текстура – це растркове зображення, що накладається на поверхню об'єкта для надання йому кольору, забарвлення або ілюзії рельєфу. У Three.js текстура (`map`) є властивістю матеріалу.

Відеотекстура – це постійно оновлювана текстура (властивість `needsUpdate` встановлена у `true`), що складається з кадрів відео. Як і звичайну текстуру, її можна накласти на будь-який об'єкт, проте для традиційного відео найбільш придатною є площа:

```
const geometry = new THREE.PlaneGeometry(1, 9/16);
const material = new THREE.MeshBasicMaterial({map: texture});
const plane = new THREE.Mesh(geometry, material);

const anchor = mindarThree.addAnchor(0);
anchor.group.add(plane);
```

Розмір площини встановлений згідно співвідношення сторін відео – для більшої точності другий параметр `PlaneGeometry` мо-

жна було встановити як співвідношення висоти та ширини відео: `video.videoHeight / video.videoWidth`.

Так само, як й для анімації моделей, доцільним є розпочинати відтворення відео при виявленні цілі та призупиняти при втраті:

```
anchor.onTargetFound = () => {
    video.play();
}
anchor.onTargetLost = () => {
    video.pause();
}
```

Якщо в одному відеофайлі міститься декілька відеофрагментів, то відтворити необхідний можна, встановивши властивість `currentTime` (час у секундах початку відтворення відео) у значення, відмінне від 0:

```
video.addEventListener("play", () => {
    video.currentTime = 0;
});
```

Результат роботи програми показано на рис. 9.2.

### 3.11 Видалення хромакею з відео

Хромакей (chroma key) традиційно використовується для поєднання двох і більше зображень або кадрів в одній композиції (сцені). Для цього одне із зображень знімається на однотонному тлі – найчастіше зеленому через те, що цифрові камери найбільш чутливі до відтінків зеленого; зображення у зеленому каналі містить менше шумів, є більш чистим та легше піддається обробці.

Видалення зеленого фону з відео, накладеного на певний об'єкт, надає можливість отримати ефекти хромакею у WebAR додатках. На жаль, простого способу видалення зеленого фону у Three.js наразі немає – необхідно заглибитися в конвеєр рендерингу WebGL.

Шейдинг – своєрідна реакція матеріалу на освітлення – передбачає використання затемнення або просвітлення окремих ділянок при створенні

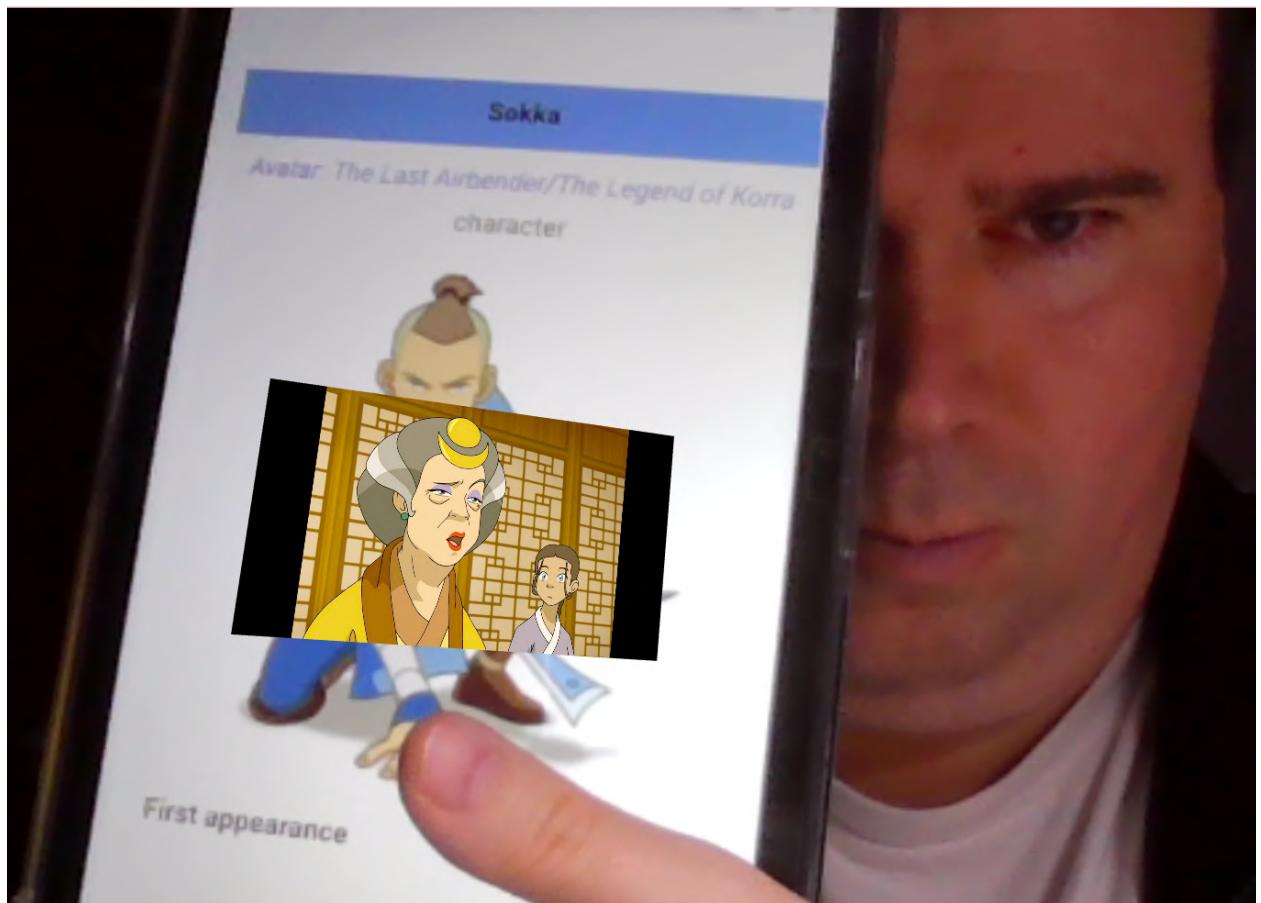


Рис. 3.20. Відтворення відео.

зображення. Ефект шейдингу заснований на сприйнятті глибини зором залежно від рівня затемнення зображення.

Шейдер – програма для шейдингу, що може включати в себе довільної складності опис поглинання та розсіювання світла, накладення текстури, відзеркалення і заломлення, затінення, зміщення поверхні і ефекти пост-обробки. Для опису шейдерів створено ряд спеціалізованих мов, орієнтованих на використання графічних процесорів.

Основні типи шейдерів:

- *вершинні шейдери* (Vertex Shader) оперують даними вершин багатогранників (координати вершини в просторі, текстурні координати, тангенс-вектор, вектор бінормалі, вектор нормалі та ін.) та можуть бути використані для видового і перспективного перетворення вершин, генерації текстурних координат, розрахунку освітлення тощо;
- *геометричні шейдери* (Geometry Shader), на відміну від вершинних, здатні працювати не лише з однією вершиною, а з усім примітивом;

- *фрагментні (піксельні) шейдери* (Pixel Shader) працюють з фрагментами зображення – пікселями, яким поставлено у відповідність деякий набір атрибутів, таких як колір, глибина, текстурні координати.

Шейдери – це дуже велика тема (але можна почати з [https://www.khronos.org/opengl/wiki/OpenGL\\_Shading\\_Language](https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language)), тому для простоти можна припустити, що шейдер – це невелика програма, яка диктує, як об'єкти рендеритимуться на виводі на полотно.

Коли у попередньому прикладі ми створювали певний матеріал (об'єкт класу `MeshBasicMaterial`), він вже використовував якийсь шейдер за замовчуванням, який просто копіював усе, що було в текстурі, на полотно:

```
const material = new THREE.MeshBasicMaterial({map: texture});
```

Ідея видалення зеленого фону полягає в тому, що кожного разу, коли ми зустрічаємо піксель, який є зеленим або досить близьким до зеленого, ми робимо цей піксель прозорим. Для цього нам необхідно написати користувацький шейдер замість стандартного.

На жаль, звичайно, шейдерні програми пишуться не на JavaScript, а на спеціальній мові GLSL (OpenGL Shading Language), код якою розміщується у файлі HTML та компілюється веб-браузером окремо від коду JavaScript.

Спочатку створимо вершинний шейдер. Для цього додамо його опис у тезі `script`, встановивши для цього значення атрибути `type` у `x-shader/x-vertex`:

```
<!-- Вершинний шейдер -->
<script type="x-shader/x-vertex" id="vertexShader">
varying mediump vec2 vUv;

void main(void)
{
    vUv = uv;
    mediump vec4 mvPosition = modelViewMatrix * vec4(position, 1.0);
    gl_Position = projectionMatrix * mvPosition;
}
</script>
```

У шейдерах є три типи змінних:

- уніформи (uniform) – це змінні, які мають однакове значення для всіх вершин (наприклад, карти освітлення, туману та тіней); доступ до уніформ може бути наданий як вершинним шейдером, так й фрагментним;
- атрибути (attribute) – це змінні, пов'язані з кожною вершиною (наприклад, положення вершини, нормаль грані та колір вершини); доступ до атрибутів можливий лише в межах вершинного шейдера;
- варіації (varying) – це змінні, які передаються від вершинного шейдера до фрагментного; для кожного фрагмента значення кожної змінної буде плавно інтерполюватися зі значень сусідніх вершин.

Вершинний шейдер запускається першим; він отримує атрибути, обчислює / маніпулює положенням кожної окремої вершини і передає додаткові дані фрагментному шейдеру. Головне, що має зробити вершинний шейдер – обчислити `gl_Position`, чотиривимірний вектор, який визначає кінцеве положення вершини на екрані перетворенням її 3D-координат на 2D-координати. Код цього шейдера відповідає стандартному (він виконує копіювання) за єдиним виключенням – збереженням у змінній `vUv` координат поточної вершини.

Другим запускається фрагментний (або піксельний) шейдер, який задає колір кожного окремого “фрагмента” (пікселя), що виводиться на екран. Фрагментний шейдер визначатиме колір встановленням `gl_FragColor` у чотиривимірний вектор, координати якого представлятимуть червоний, зелений, жовтий і синій кольори відповідно:

```
<!-- Фрагментний шейдер -->
<script type="x-shader/x-fragment" id="fragmentShader">
uniform mediump sampler2D mytexture;
uniform mediump vec3 color;
varying mediump vec2 vUv;

void main(void)
{
```

```

    mediump vec3 tColor = texture2D( mytexture, vUv ).rgb;
    mediump float a = (length(tColor - color) - 0.7) * 7.0;
    gl_FragColor = vec4(tColor, a);
}
</script>

```

Тут `mytexture` – текстура, що опрацьовується, `vUv` – передані з вершинного шейдера координати точки, колір якої визначається, `color` – колір, що ми його намагатимемось видалити. Колір точки текстири `mytexture` за координатами `vUv` визначається функцією `texture2D` у форматі RGB.

Внесемо деякі зміни до прикладу відтворення відео. Насамперед нам знадобиться відео – наприклад, в <https://www.youtube.com/watch?v=0Th0ce5aXxU> власне ефект зеленого відео розпочинається із 6 секунди, в яке й встановлене значення `currentTime`:

```

const video = await loadVideo("white smoke green screen "+
  "_ white drop green screen.mp4");
const texture = new THREE.VideoTexture(video);

video.addEventListener("play", () => {
  video.currentTime = 6;
});

const geometry = new THREE.PlaneGeometry(1,
  video.videoHeight / video.videoWidth);

```

`ShaderMaterial` надає створити спеціальний матеріал – користувальський шейдер – за допомогою вершинного та фрагментного шейдерів. При використанні `ShaderMaterial` слід мати на увазі, що він буде коректно рендеритися тільки `WebGLRenderer`, оскільки GLSL код у властивостях `vertexShader` і `fragmentShader` повинен бути скомпільований і запущений на GPU за допомогою WebGL.

```

let customUniforms = {
  mytexture: {
    type: "t",

```

```

    value: texture
},
color: {
  type: "c",
  value: new THREE.Color(0x00ff00)
}
};

const material = new THREE.ShaderMaterial({
uniforms: customUniforms,
vertexShader:
  document.getElementById("vertexShader").textContent,
fragmentShader:
  document.getElementById("fragmentShader").textContent,
transparent: true
});

```

`customUniforms` – користувацька уніформа, що передається до користувацького шейдеру та містить відеотекстуру `texture` та колір `color`. Хоча тут він заданий як `Three.Color(0x00ff00)` – завжди зелений колір, замість нього може бути вказаний будь-який бажаний колір фону, що його необхідно видалити.

`vertexShader` та `fragmentShader` містять код шейдерів, отримуваний з документу HTML, а встановлений у `true` параметр `transparent` вказує, що користувацький матеріал буде прозорим.

Таким чином, наша площа буде рендеритися за допомогою користувацького матеріалу:

```
const plane = new THREE.Mesh(geometry, material);
```

Продемонструємо деякі налаштування її геометрії (рис. 3.21):

```

plane.rotation.x = Math.PI / 2; // поворот навколо вісі x
plane.position.y = 0.7;          // зміщення вгору
plane.scale.multiplyScalar(4); // масштабування

```



Рис. 3.21. Демонстрація роботи хромакею.

На рис. 3.21 можна побачити залишки зеленого кольору – це дає привід поекспериментувати із кодом фрагментного шейдеру.

Для зручності використання доцільно оформити створення користувальського шейдеру з ефектом хромакею у вигляді модуля `chroma-video.js`:

```
import * as THREE
from "https://unpkg.com/three/build/three.module.js";

export const createChromaMaterial = (texture, keyColor) => {
  const keyColorObject = new THREE.Color(keyColor);
  const material = new THREE.ShaderMaterial({
    uniforms: {
```

```

mytexture: {
    type: "t",
    value: texture
},
color: {
    type: "c",
    value: keyColorObject
}
},
vertexShader:
"varying mediump vec2 vUv;\n" +
"void main(void)\n" +
"{\n" +
"    vUv = uv;\n" +
"    mediump vec4 mvPosition = modelViewMatrix *" +
"        vec4( position, 1.0 );\n" +
"    gl_Position = projectionMatrix * mvPosition;\n" +
"}",
fragmentShader:
"uniform mediump sampler2D mytexture;\n" +
"uniform mediump vec3 color;\n" +
"varying mediump vec2 vUv;\n" +
"void main(void)\n" +
"{\n" +
"    mediump vec3 tColor = texture2D( mytexture, vUv ).rgb;\n" +
"    mediump float a = (length(tColor - color) - 0.7) * 7.0;\n" +
"    gl_FragColor = vec4(tColor, a);\n" +
"}",
transparent: true
);
return material;
}

```

Код шейдерів у модулі є включеним як текст мовою GLSL, тому створювати відповідні блоки для них у документі HTML більше потреби немає.

Функція `createChromaMaterial` створює користувацький матеріал що містить текстуру `texture` з видаленим кольором `keyColor`. На початку програми її необхідно імпортувати:

```
import {createChromaMaterial} from "chroma-video.js";
```

Створення користувацького матеріалу за допомогою `createChromaMaterial` для видалення зеленого кольору:

```
const material = createChromaMaterial(texture, 0x00ff00);
```

### 3.12 CSS-рендеринг

Розглянемо незвичну, але потужну техніку, що може бути використана у WebAR додатках – використання елементів HTML та CSS для AR-контенту, таких як `div`, `image`, `p`, може бути прикріплений до цільових якорів.

```
<html>
  <head>
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0">
    <script type="importmap">
    {
      "imports": {
        "three": "https://unpkg.com/three/build/three.module.js"
      }
    }
    </script>
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
    ></script>
    <script src=".main.js" type="module"></script>

    <style>
      html, body {
```

```

        position: relative;
        margin: 0;
        width: 100%;
        height: 100%;
        overflow: hidden
    }
    #ar-div {
        width: 1000px;
        height: 1000px;
        visibility: hidden;
        background: #0000ff;
        opacity: 0.6;
        font-size: 100px;
        display: flex;
        justify-content: center;
        align-items: center;
        color: white;
    }

```

</style>

</head>

<body>

<div id="ar-div">

CSS div як елемент доповненої реальності

</div>

</body>

</html>

div є частиною документа HTML – контейнером, що описується стилем CSS та може містити що завгодно. У MindAR для об'єктів Three.js значення 1 відповідає найбільшому розміру цільового зображення, проте для об'єктів CSS це значення – 1000, тому для контейнеру ar-div встановлені такі стилеві налаштування: ширина (width) та висота (height) – 1000 пікселів (1000px), при першому зверненні контейнер є невидимим (visibility: hidden), колір фону (background) – синій (#0000ff), колір

елементів контейнеру (`color`) – білий (`white`), прозорість (`opacity`) – 60% (0.6), розмір шрифту надписів (`font-size`) – 100 пікселів, розташування елементів у контейнері (`display`) динамічне (`flex`), елементи контейнеру розташовуються (`justify-content`), починаючи з центру (`center`), та вирівнюються (`align-items`) за центром.

Three.js та MindAR мають можливість рендерингу HTML та елементів CSS, проте у MindAR вона є вбудованою, у той час як у Three.js необхідно підключити додатковий модуль `CSS3DRenderer`.

`CSS3DRenderer` можна використовувати для ієрархічних 3D-перетворень DOM-елементів через властивість CSS3 `transform`. Цей рендерер особливо цікавий, якщо ви хочете застосувати 3D-екти на сайті без рендеринга на основі полотна. Він також може бути використаний для того, щоб об'єднати DOM-елементи з WebGL-контентом.

Однак із `CSS3DRenderer` неможливо використовувати систему матеріалів та геометрії Three.js, тому що `CSS3DRenderer` орієнтований на звичайні DOM-елементи. Ці елементи загортуються в спеціальні об'єкти (`CSS3DObject` або `CSS3DSprite`) і потім додаються до сцени.

`CSS3DObject` не є частиною Three.js API, тому доводиться імпортувати його явно:

```
import {CSS3DObject} from "https://unpkg.com/three/examples/"+  
"jsm/renderers/CSS3DRenderer.js";  
  
const THREE = window.MINDAR.IMAGE.THREE;  
  
document.addEventListener("DOMContentLoaded", () => {  
    const start = async() => {  
        const mindarThree = new window.MINDAR.IMAGE.MindARThree({  
            container: document.body,  
            imageTargetSrc: "altabor.mind",  
        });  
  
        const {renderer, cssRenderer, scene, cssScene, camera} =  
            mindarThree;
```

```

const obj = new CSS3DObject(document.querySelector("#ar-div"));
const cssAnchor = mindarThree.addCSSAnchor(0);
cssAnchor.group.add(obj);

await mindarThree.start();
renderer.setAnimationLoop(() => {
    cssRenderer.render(cssScene, camera);
});
start();
);

```

Із об'єкту `mindarThree` можна отримати як звичні об'єкти `renderer` та `scene` класів `WebGLRenderer` та `Scene`, так їй `cssRenderer` – об'єкт класу `CSS3DRenderer`, та `cssScene` – ще один об'єкт класу `Scene`.

`obj` є об'єктом класу `CSS3DObject`, що викликом `document.querySelector` пов'язується із визначеним у документі HTML контейнером `ar-div`. Створення цього об'єкту замінює раніше використовувану тріаду “геометрія + матеріал = об'єктна сітка”.

CSS об'єкт `obj` зв'язується зі спеціальним типом якоря – якорем CSS, що створюється викликом `addCSSAnchor`.

Нарешті, у циклі анімації метод `render` об'єкту `cssRenderer` використовується для того, щоб відобразити CSS-сцену (`cssScene`) з точки зору камери (`camera`) (рис. 3.22).

### 3.13 Відтворення потокового відео з Vimeo

Для відтворення відео засобами HTML5 воно повинно мати джерелом відеофайл – так, для використання відео з відеосервісів, таких як Vimeo та YouTube, відеоролики доводилось завантажувати. Можливість використання потокового відео є дуже корисною, і, якщо певний відеосервіс надає можливість убудування фрейму з відео у документ HTML, його можна було б використати в якості об'єкту доповненої реальності через CSS-рендеринг.

HTML-елемент `iframe` є способом убудувати іншу HTML-сторінку в посточну – продемонструємо це, внісши зміни до попереднього коду документу



Рис. 3.22. CSS-рендеринг.

HTML.

Спочатку спростимо стиль `ar-div`, вказавши лише, що при створенні даний контейнер є невидимим:

```
#ar-div {  
    visibility: hidden;  
}
```

Далі вкажемо код програвача Vimeo для певного відео – наприклад, для <https://vimeo.com/246615602> він буде <https://player.vimeo.com/video/246615602>. Параметрами `iframe` є також ширина і висота – вони встановлені у значення 1000 пікселів, що відповідає однічному розміру цільового зображення:

```
<div id="ar-div">  
    <iframe src="https://player.vimeo.com/video/246615602"  
        width="1000" height="1000" frameborder="0"></iframe>  
</div>
```

У результаті отримаємо накладене на цільове зображення вікно програвача Vimeo (рис. 3.23).

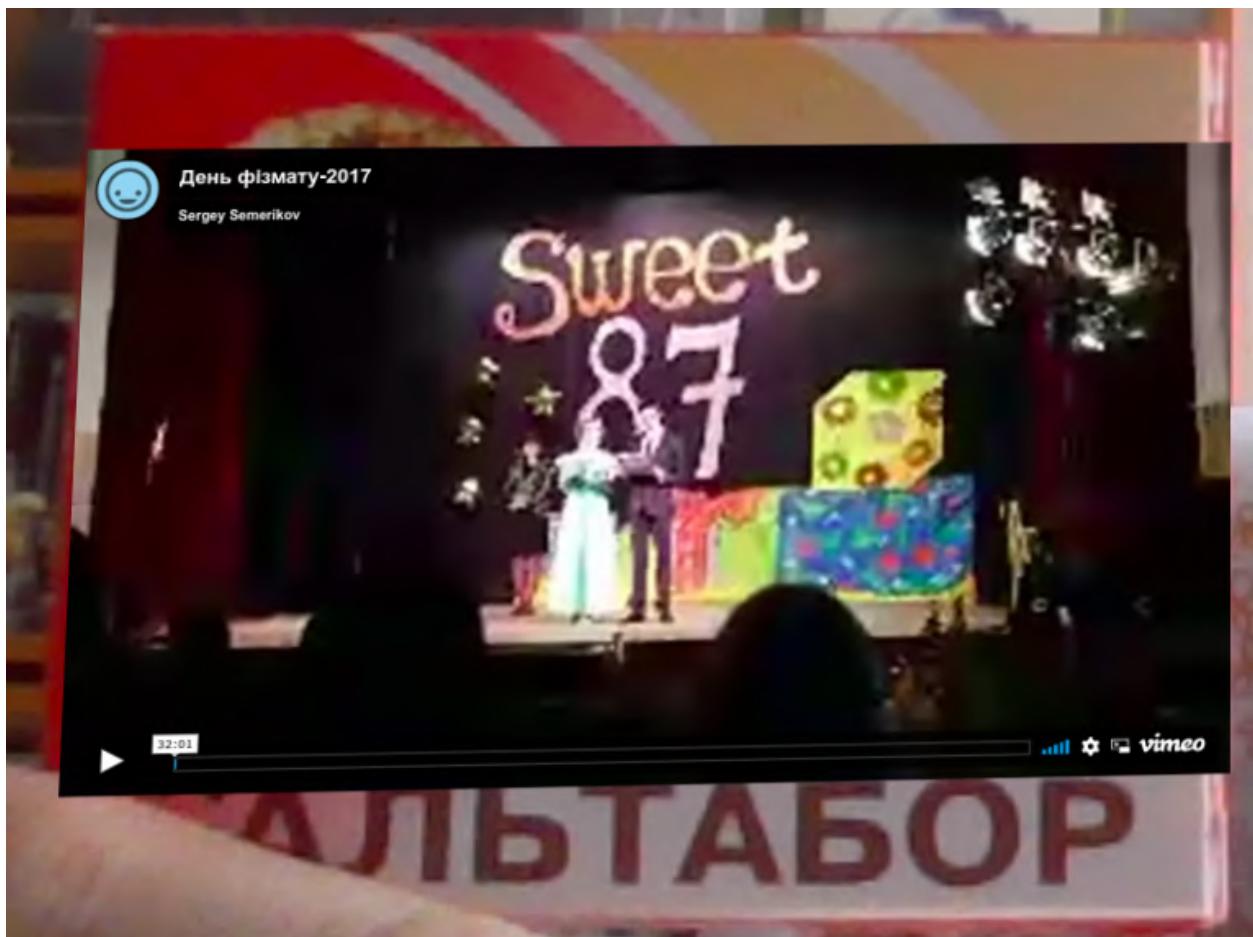


Рис. 3.23. Відео з потокового сервісу Vimeo.

Просто, але таке відео не дуже кероване – щоб розпочати його відтворення, необхідно натиснути на відповідну кнопку програвача, а при втраті цільового зображення відео продовжує відтворюватись замість призупинення (їого не буде видно, проте звук транслюватиметься).

Для управління скористаємось Vimeo Player SDK (<https://developer.vimeo.com/player/sdk>, простір імен Vimeo) – це потребує додавання до документу HTML команди підключення відповідної бібліотеки:

```
<script src="https://player.vimeo.com/api/player.js"></script>
```

До `main.js` додамо код для створення об'єкту `video` класу `Player`, а також обробку подій `onTargetFound` і `onTargetLost` об'єкту `cssAnchor` і події `play` об'єкту `video`:

```
var video = new Vimeo.Player(document.querySelector("iframe"));

//video.setLoop(true);
```

```

cssAnchor.onTargetFound = () => {
    video.play().catch(() => {});
}

cssAnchor.onTargetLost = () => {
    video.pause();
}

video.on("play", () => {
    video.setCurrentTime(0);
});

```

Методи для управління програвачем Vimeo реалізовані як проміси, тому до обробника `onTargetFound` додана порожня функція опрацювання помилки, яка виникає через те, що відео автоматично не відтворюється (задля уникнення описаних вище проблем із автовідтворенням відео та аудіо у веб-браузері Safari). Лише після того, як користувач явно запустить відео, метод `play` буде працювати правильно.

При втраті цілі відео призупиняється – за необхідності, після цього можна “перекрутити” відео на початок викликом `setCurrentTime(0)`.

Нарешті, за потреби виклик `setLoop` з параметром `true` намагатиметься закільцовувати відео.

### 3.14 Відтворення потокового відео з YouTube

Лобова спроба отримати некероване відео з YouTube, скоріше за все, буде невдалою або через заборону власника відео його убудовувати, або через неможливість запуску користувачем натисканням на кнопку.

Насправді відео запускається, але тут же зупиняється через зміну координат `iframe`. AR-рушій безперервно оновлює положення елемента та властивість CSS `transform` – ймовірно, саме змінна останньої призводить до зупинки програвача.

Тому використання YouTube IFrame API ([https://developers.google.com/youtube/iframe\\_api\\_reference](https://developers.google.com/youtube/iframe_api_reference), простір імен YT) є наразі

єдиним способом створення керованого відео у доповненій реальності.

Зміст документу HTML дещо відрізняється від попереднього прикладу: до контейнеру `ar-div` замість `iframe` включено контейнер `player` – це дозволить застосувати виклик `document.querySelector` не до `iframe` (який може у документі бути не один), а до ідентифікованого своїм іменем елемента `player`. Водночас налаштування стилю зроблені для всіх елементів `iframe` – вибір ширини (1000 пікселів) обґрунтований раніше, а висота пропорційна відповідному розміру цільового зображення:

```
<html>
  <head>
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0">
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
      ></script>
    <script type="importmap">
    {
      "imports": {
        "three": "https://unpkg.com/three/build/three.module.js"
      }
    }
    </script>
    <script src="./main.js" type="module"></script>

  <style>
    html, body {
      position: relative;
      margin: 0;
      width: 100%;
      height: 100%;
      overflow: hidden
    }
    #ar-div {
      visibility: hidden;
```

```

        }
    
```

```

        iframe {
            width: 1000px;
            height: 750px;
        }
    
```

```

    </style>

```

```

</head>

```

```

<body>
    <div id="ar-div">
        <div id="player"/>
    </div>
</body>

```

```

</html>

```

Зверніть увагу на те, що у документі HTML відсутня традиційна команда підключення необхідної бібліотеки:

```
<script src = "https://www.youtube.com/iframe_api"></script>
```

Власне, будь-який елемент документу HTML, і `script` не є виключенням, можна створити безпосередньо у коді JavaScript, що й продемонстровано у файлі `main.js`, початок якого є традиційними:

```
import {CSS3DObject} from
"https://unpkg.com/three/examples/jsm/renderers/CSS3DRenderer.js";
```

```
const THREE = window.MINDAR.IMAGE.THREE;
```

Завантажити відео з YouTube можна за його ідентифікатором або гіперпосиланням, проте створити програвач – лише за ідентифікатором. Функція `getVideoId` намагається перетворити гіперпосилання на ідентифікатор відео:

```
function getVideoId(url) {
    const urltypes = [
        "https://www.youtube.com/shorts/",

```

```

        "https://youtu.be/",
        "https://www.youtube.com/watch?v="
    ];

    if(url.startsWith(urltypes[0]))
        return url.substr(urltypes[0].length)
    if(url.startsWith(urltypes[1]))
        return url.substr(urltypes[1].length)
    if(url.startsWith(urltypes[2])) {
        let s = url.substr(urltypes[2].length);
        let index = s.indexOf("&");
        return (index === -1) ? s : s.substr(0, index);
    }
    return url;
}

```

Функція `createYoutube` повертає проміс, а тому, як асинхронна функція, має використовуватись із викликом `await`. У промісі спочатку створюється елемент `script`, який й відповідає за підключення бібліотеки `https://www.youtube.com/iframe_api`. Його додавання у документі HTML виконується в чітко визначене місце – перед першим елементом `script` у документі:

```

const createYoutube = (url) => {
    return new Promise((resolve, reject) => {
        var tag = document.createElement("script");
        tag.src = "https://www.youtube.com/iframe_api";
        var firstScriptTag = document.getElementsByTagName("script")[0];
        firstScriptTag.parentNode.insertBefore(tag, firstScriptTag);

        const onYouTubeIframeAPIReady = () => {
            const player = new YT.Player("player", {
                videoId: getVideoId(url),
                events: {
                    onReady: () => {

```

```

        resolve(player);
    }
}
}) ;
}

window.onYouTubeIframeAPIReady = onYouTubeIframeAPIReady;
}) ;
}

```

Будь-яка веб-сторінка, яка використовує API IFrame, повинна також реалізувати функцію `onYouTubeIframeAPIReady` – API буде викликати цю функцію, коли сторінка закінчить завантаження JavaScript для API програвача, що дозволить нам потім використовувати API на нашій сторінці. Таким чином, ця функція може створювати об'єкти програвача, які ви відображатимуться при завантаженні сторінки.

У конструкторі відеопрограма `Player` вказуються наступні параметри:

1. Перший параметр вказує або DOM-елемент, або ідентифікатор HTML-елемента, куди API вставить тег `iframe`, що містить програвач. API IFrame замінить елемент `player` на елемент `iframe`, що містить програвач.
2. Другий параметр – це об'єкт, який задає параметри програвача. Об'єкт містить наступні властивості:
  - `width` – ширина відеопрограма (значення за замовчуванням – 640);
  - `height` – висота відеопрограма (значення за замовчуванням – 390);
  - `videoId` – ідентифікатор відео на YouTube, який ідентифікує відео, що буде завантажуватися програвачом;
  - `playerVars` – властивості об'єкта, що визначають параметри програвача, які можуть бути використані для його налаштування;

- **events** – властивості об'єкту, що визначають події, які генерує API, та функції (слухачі подій), які API буде викликати при настанні цих подій. У прикладі конструктор вказує, що об'єкт **player** буде повертатись при виникненні події **onReady**.

Після завантаження документу виклик **createYoutube** для заданого своїм URL відео повертає об'єкт **player** для управління програвачем, що демонструється на прикладах перемотки (**seekTo**), а також програвання (**playVideo**) та призупинення (**pauseVideo**) відео за подіями виявлення та втрати цільового зображення (рис. 3.24):

```
document.addEventListener("DOMContentLoaded", () => {
  const start = async() => {
    const player = await
      createYoutube("https://www.youtube.com/watch?v=ZvMHF3zBuis");

    const mindarThree = new window.MINDAR.IMAGE.MindARThree({
      container: document.body,
      imageTargetSrc: "altabor.mind",
    });
    const {renderer, cssRenderer, scene, cssScene, camera} =
      mindarThree;

    const obj = new CSS3DObject(document.querySelector("#ar-div"));
    const cssAnchor = mindarThree.addCSSAnchor(0);
    cssAnchor.group.add(obj);

    player.seekTo(0);

    cssAnchor.onTargetFound = () => {
      player.playVideo();
    }
    cssAnchor.onTargetLost = () => {
      player.pauseVideo();
    }
  }
})
```

```
await mindarThree.start();
renderer.setAnimationLoop(() => {
    cssRenderer.render(cssScene, camera);
});
start();
});
```



Рис. 3.24. Відео з потокового сервісу YouTube.

Vimeo API та YouTube IFrame API є потужним засобом управління відео, який доцільно застосовувати для великих відео. Для відео малого розміру та відео для хромакею доцільним є застосування відеотекстур Three.js. Вибір, який метод застосовувати, значною мірою залежить від пропускної здатності мережі, у якій працюватиме WebAR додаток.

### 3.15 Приклад: візитівка у доповненій реальності

Доповнені візитівки широко застосовуються – за пошуковим запитом “augmented reality business card” можна отримати велику кількість ідей для створення власної візитівки, яку можна перетворити на своєрідне онлайн-портфоліо.

Цільовим зображенням може бути традиційна візитівка (рис. 3.25) або довільне зображення, що може бути використане для власної ідентифікації.

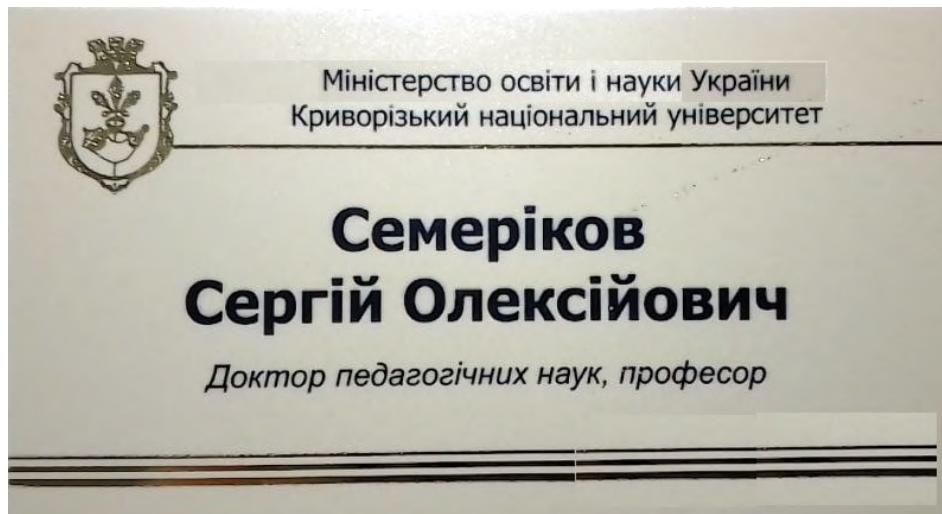


Рис. 3.25. Візитівка як цільове зображення.

Його доповненням можуть бути 3D-моделі, відео, аудіо, зображення або текст, анімації тощо – можна створити ефект переходу за допомогою вичвітання або просто анімувати кнопки. Головне – не показувати весь вміст відразу і робити його динамічним.

Необхідно також налаштувати взаємодію з елементами візитівки, передбачивши різну реакцію на подію натискання миші – запуск відео, відображення додаткової інформацію, відкриття зовнішньої веб-сторінку тощо. Для мобільних пристройів можна було б спробувати жести змахування вліво або вправо.

Це не потребуватиме створення елементів у документі HTML – все, що необхідно, може бути створене засобами JavaScript:

```
<html>
  <head>
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
```

```

<script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
></script>
<script type="importmap">
{
  "imports": {
    "three": "https://unpkg.com/three/build/three.module.js"
  }
}
</script>
<script src="./main.js" type="module"></script>
<style>
  html, body {position: relative; margin: 0; width: 100%; height:
</style>
</head>
<body>
</body>
</html>

```

Для спрощення завантаження текстур додовнімо бібліотеку loader.js асинхронними функціями `loadTexture` та `loadTextures`, що повертають проміси з однією текстурою та масивом текстур відповідно:

```

export const loadTexture = (path) => {
  return new Promise((resolve, reject) => {
    const loader = new THREE.TextureLoader();
    loader.load(path, (texture) => {
      resolve(texture);
    });
  });
}

export const loadTextures = (paths) => {
  const loader = new THREE.TextureLoader();
  const promises = [];
  for (let i = 0; i < paths.length; i++) {

```

```

        promises.push(new Promise((resolve, reject) => {
            loader.load(paths[i], (texture) => {
                resolve(texture);
            });
        }));
    }
    return Promise.all(promises);
}

```

Спробуємо спільно використати рендеринг як на полотні, так й на елементі CSS:

```

import {CSS3DObject} from
    "https://unpkg.com/three/examples/jsm/renderers/CSS3DRenderer.js";
import {loadGLTF, loadTexture, loadTextures, loadVideo} from
    "./loader.js";
const THREE = window.MINDAR.IMAGE.THREE;

```

Застосуємо компілятор зображень до зображення візитівки для отримання файлу опорних точок cc\_card.mind:

```

document.addEventListener("DOMContentLoaded", () => {
    const start = async() => {

        const mindarThree = new window.MINDAR.IMAGE.MindARThree({
            container: document.body,
            imageTargetSrc: "cc_card.mind",
        });
    }
}

```

Обидва (звичайний та CSS) рендерери, сцени та камеру отримаємо з бібліотеки MindAR:

```

const {renderer, cssRenderer, scene, cssScene, camera} =
    mindarThree;

```

Для того, щоб зробити видимими моделі, застосуємо напівсферичне освітлення:

```
const light = new THREE.HemisphereLight( 0xfffffff, 0xb0b0b0, 1 );
scene.add(light);
```

Завантажимо необхідні для роботи текстури, локально або за URL:

```
const [
    cardTexture,
    emailTexture,
    locationTexture,
    webTexture,
    profileTexture,
    leftTexture,
    rightTexture,
    portfolioItem0Texture,
    portfolioItem1Texture,
    portfolioItem2Texture,
] = await loadTextures([
    "cc_card.png",
    "https://live.staticflickr.com/1316/1301796545_7aee4aef61.jpg",
    "https://live.staticflickr.com/65535/48078008041_21e1cc1a02_b.jpg",
    "https://live.staticflickr.com/4845/44484472720_e1d963af49_b.jpg",
    "https://live.staticflickr.com/4540/37599867624_db18a44d8c.jpg",
    "https://live.staticflickr.com/8579/15599087074_ac871542f3_w.jpg",
    "https://live.staticflickr.com/8667/16221493785_734a8ac248_w.jpg",
    "https://live.staticflickr.com/4285/34414016404_424abde5f9_b.jpg",
    "ETQ_logo_Pinchuk.png",
    "CTE_big.png",
]);
```

Локальні файли cc\_card.png, ETQ\_logo\_Pinchuk.png, CTE\_big.png – зображення візитівки й логотипи журналів *Educational Technology Quarterly* та *CTE Workshop Proceedigs*.

Розміри площини, текстурою для якої є файл cc\_card.png, дібраці відповідно до співвідношення сторін візитівки (але можна було б отримати відповідні розміри з властивостей текстури й застосувати формулу замість сталого значення 0.552):

```

const planeGeometry=new THREE.PlaneGeometry(1, 0.552);
const cardMaterial=new THREE.MeshBasicMaterial({map:cardTexture});
const card=new THREE.Mesh(planeGeometry, cardMaterial);

```

Усі іконки матимуть одну й ту саму колову геометрію, але різні текстури:

```

const iconGeometry=new THREE.CircleGeometry(0.075, 32);
const emailMaterial=new
    THREE.MeshBasicMaterial({map:emailTexture});
const webMaterial=new THREE.MeshBasicMaterial({map:webTexture});
const profileMaterial=new
    THREE.MeshBasicMaterial({map:profileTexture});
const locationMaterial=new
    THREE.MeshBasicMaterial({map:locationTexture});
const leftMaterial=new THREE.MeshBasicMaterial({map:leftTexture});
const rightMaterial=new
    THREE.MeshBasicMaterial({map:rightTexture});
const emailIcon=new THREE.Mesh(iconGeometry, emailMaterial);
const webIcon=new THREE.Mesh(iconGeometry, webMaterial);
const profileIcon=new THREE.Mesh(iconGeometry, profileMaterial);
const locationIcon=new THREE.Mesh(iconGeometry, locationMaterial);
const leftIcon=new THREE.Mesh(iconGeometry, leftMaterial);
const rightIcon=new THREE.Mesh(iconGeometry, rightMaterial);

```

Спочатку зробимо портфоліо з трьох елементів, причому для першого створимо дві площини – з відеотекстурою <https://github.com/mrdoob/three.js/blob/dev/examples/textures/sintel.mp4> та її попереднім переглядом: це надасть можливість продемонструвати ефект “заміни” зображення на відео:

```

const portfolioItem0Video = await loadVideo("sintel.mp4");
portfolioItem0Video.muted = true;
const portfolioItem0VideoTexture = new
    THREE.VideoTexture(portfolioItem0Video);
const portfolioItem0VideoMaterial = new
    THREE.MeshBasicMaterial({map: portfolioItem0VideoTexture});

```

```

const portfolioItem0Material = new
    THREE.MeshBasicMaterial({map: portfolioItem0Texture});
const portfolioItem1Material = new
    THREE.MeshBasicMaterial({map: portfolioItem1Texture});
const portfolioItem2Material = new
    THREE.MeshBasicMaterial({map: portfolioItem2Texture});

const portfolioItem0V = new
    THREE.Mesh(planeGeometry, portfolioItem0VideoMaterial);
const portfolioItem0 = new
    THREE.Mesh(planeGeometry, portfolioItem0Material);
const portfolioItem1 = new
    THREE.Mesh(planeGeometry, portfolioItem1Material);
const portfolioItem2 = new
    THREE.Mesh(planeGeometry, portfolioItem2Material);

```

Виходячи з того, що центр цільового зображення ширину 1 відповідає початку координат, налаштуємо координати іконок знизу:

```

profileIcon.position.set(-0.42, -0.5, 0);
webIcon.position.set(-0.14, -0.5, 0);
emailIcon.position.set(0.14, -0.5, 0);
locationIcon.position.set(0.42, -0.5, 0);

```

Візуальні елементи портфоліо (центральне зображення та стрілки поряд із ним) згрупуємо та розташуємо вище іконок:

```

const portfolioGroup = new THREE.Group();
portfolioGroup.position.set(0, 0, -0.01);
portfolioGroup.position.set(0, 0.6, -0.01);
portfolioGroup.add(portfolioItem0);
portfolioGroup.add(leftIcon);
portfolioGroup.add(rightIcon);
leftIcon.position.set(-0.7, 0, 0);
rightIcon.position.set(0.7, 0, 0);

```

Завантажимо модель та розташуємо її на сцені. Розміри та координати добираються відповідно використаній моделі:

```

const avatar = await
loadGLTF("https://raw.githubusercontent.com/AR-js-org/" +
"AR.js/master/aframe/examples/image-tracking/nft/trex/" +
"scene.gltf");
avatar.scene.scale.set(0.02, 0.02, 0.02);
avatar.scene.rotation.set(0, 11*Math.PI/180, 0);
avatar.scene.position.set(-0.40, -0.45, -0.25);

```

Створимо якір, пов'язаний із цільовим зображенням, та прив'яжемо до нього модель, візитівку, іконки та групу візуальних елементів портфолію:

```

const anchor = mindarThree.addAnchor(0);
anchor.group.add(avatar.scene);
anchor.group.add(card);
anchor.group.add(emailIcon);
anchor.group.add(webIcon);
anchor.group.add(profileIcon);
anchor.group.add(locationIcon);
anchor.group.add(portfolioGroup);

```

Створимо контейнер CSS, який розташуємо під візитівкою (ширина по-лотна – 1000 пікселів HTML):

```

const textElement = document.createElement("div");
const textObj = new CSS3DObject(textElement);
textObj.position.set(0, -1000, 0);
textObj.visible = false;
textElement.style.background = "#FFFFFF";
textElement.style.padding = "30px";
textElement.style.fontSize = "60px";

```

У контейнері може бути розміщено будь-що – ми використаємо його для тексту, який прив'яжемо до CSS-якоря:

```

const cssAnchor = mindarThree.addCSSAnchor(0);
cssAnchor.group.add(textObj);

```

До всіх елементів доповненої візитівки, що будуть реагувати на натискання кнопки миші, додамо відповідний атрибут – інші опрацьовуватись не будуть:

```
leftIcon.userData.clickable = true;
rightIcon.userData.clickable = true;
emailIcon.userData.clickable = true;
webIcon.userData.clickable = true;
profileIcon.userData.clickable = true;
locationIcon.userData.clickable = true;
portfolioItem0.userData.clickable = true;
portfolioItem0V.userData.clickable = true;
portfolioItem1.userData.clickable = true;
portfolioItem2.userData.clickable = true;
```

Створимо масив елементів портфоліо та встановимо, що поточним є перший з них:

```
const portfolioItems = [portfolioItem0,
    portfolioItem1,
    portfolioItem2];
let currentPortfolio = 0;
```

За подією натискання кнопки миші визначимо нормалізовані координати на площині полотна та за допомогою методу кидання променів визначимо перелік об'єктів Three.js, на яких ми могли б натиснути кнопку:

```
document.body.addEventListener("click", (e) => {
    const mouseX = (e.clientX / window.innerWidth) * 2 - 1;
    const mouseY = -(e.clientY / window.innerHeight) * 2 + 1;
    const mouse = new THREE.Vector2(mouseX, mouseY);
    const raycaster = new THREE.Raycaster();
    raycaster.setFromCamera(mouse, camera);
    const intersects =
        raycaster.intersectObjects(scene.children, true);
```

Якщо перелік перетинів є непорожнім, візьмемо перший з них (як найближчий) та визначимо, якому об'єкту він належить:

```

if (intersects.length > 0) {
    let o = intersects[0].object;
    while (o.parent && !o.userData.clickable)
        o = o.parent;

```

Якщо об'єкт, на який потрапила миша, відноситься до тих, що реагує, визначаємо конкретну реакцію:

```
if (o.userData.clickable) {
```

При натисканні на стрілки вліво чи вправо визначаємо новий поточний елемент портфоліо, призупиняємо пов'язане з першим елементом відео, видаємо з групи візуальних елементів портфоліо старий поточний елемент та додаємо новий:

```

if (o === leftIcon || o === rightIcon) {
    if (o === leftIcon)
        currentPortfolio = (currentPortfolio - 1 +
            portfolioItems.length) % portfolioItems.length;
    else
        currentPortfolio = (currentPortfolio + 1) %
            portfolioItems.length;
    portfolioItem0Video.pause();
    for (let i = 0; i < portfolioItems.length; i++)
        portfolioGroup.remove(portfolioItems[i]);
    portfolioGroup.add(portfolioItems[currentPortfolio]);
}

```

При натисканні на зображення для попереднього перегляду першого елементу портфоліо замінюємо його в групі візуальних елементів на відео, яке й запускаємо:

```

else if (o === portfolioItem0) {
    portfolioGroup.remove(portfolioItem0);
    portfolioGroup.add(portfolioItem0V);
    portfolioItems[0] = portfolioItem0V;
    portfolioItem0Video.play();
}

```

При натисканні на відео призупиняємо чи відновлюємо програвання у залежності від його поточного стану:

```
else if (o === portfolioItem0V) {  
    if (portfolioItem0Video.paused) {  
        portfolioItem0Video.play();  
    } else {  
        portfolioItem0Video.pause();  
    }  
}
```

При натисканні на іконки візуалізуємо контейнер CSS з відповідним написом:

```
else if (o === webIcon) {  
    textObj.visible = true;  
    textElement.innerHTML = "https://kdpu.edu.ua/semerikov";  
} else if (o === emailIcon) {  
    textObj.visible = true;  
    textElement.innerHTML = "semerikov [at] gmail.com";  
} else if (o === profileIcon) {  
    textObj.visible = true;  
    textElement.innerHTML =  
        "https://www.researchgate.net/profile/Serhiy-Semerikov"  
} else if (o === locationIcon) {  
    textObj.visible = true;  
    textElement.innerHTML = "Kryvyi Rih, Ukraine";  
}
```

При натисканні на другий та третій елементи портфоліо відкриваємо у браузері сайти:

```
else if (o == portfolioItem1) {  
    window.open("https://acnsci.org/journal");  
} else if (o == portfolioItem2) {  
    window.open("https://acnsci.org/cte");  
}
```

```
    }  
}  
});
```

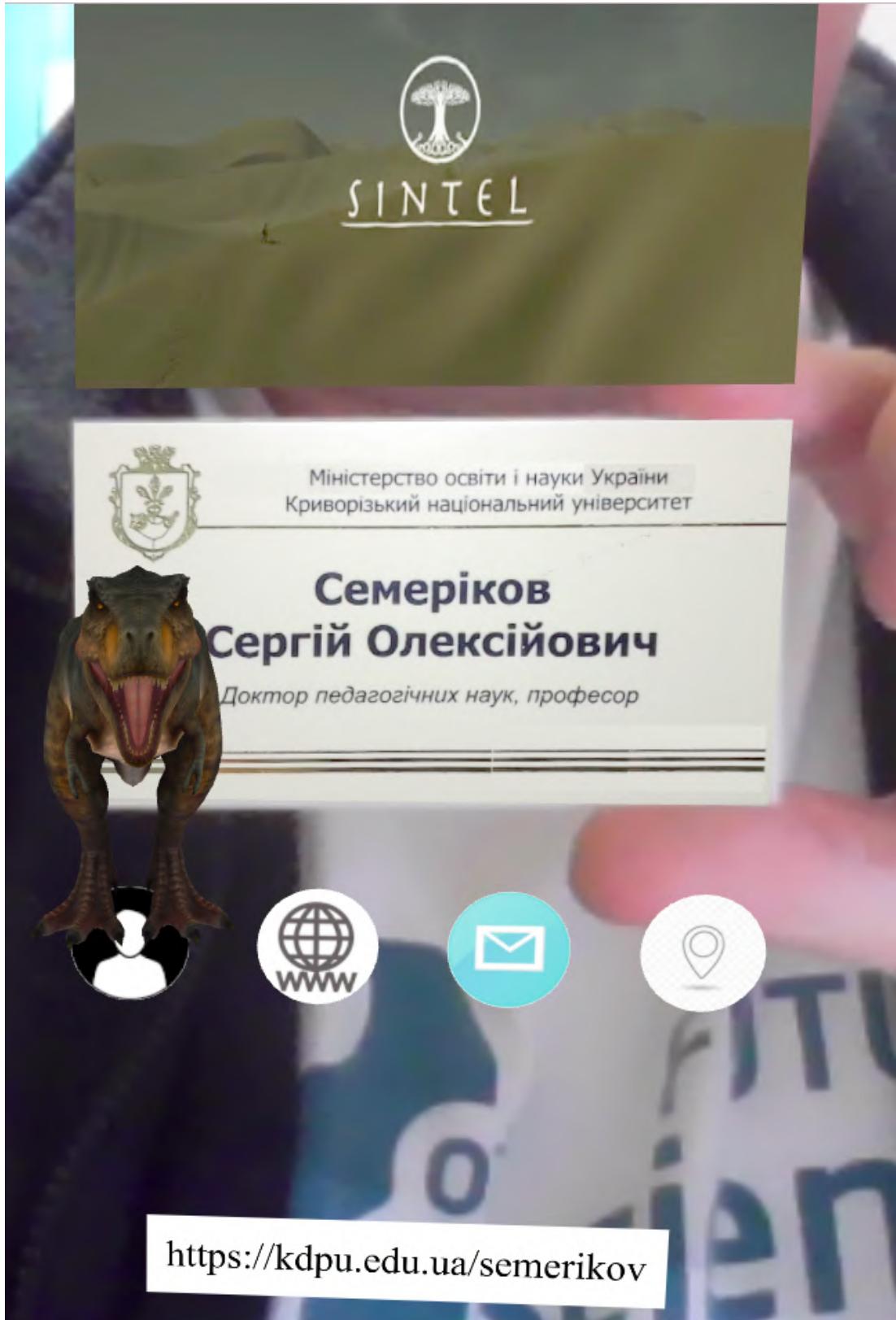


Рис. 3.26. Доповнена візитівка.

У залежності від поточного часу циклічно змінюємо розмір іконок та координати моделі:

```
const clock = new THREE.Clock();
await mindarThree.start();
renderer.setAnimationLoop(() => {
    const delta = clock.getDelta();
    const elapsed = clock.getElapsedTime();
    const iconScale = 1 + 0.2 * Math.sin(elapsed*5);
    [webIcon, emailIcon, profileIcon, locationIcon].forEach((icon) =>
        icon.scale.set(iconScale, iconScale, iconScale);
    });

    avatar.scene.position.set(-0.40,-0.45,-0.25+Math.sin(elapsed));
```

Нарешті, виконуємо рендеринг обох сцен – звичайної та CSS (рис. 3.26):

```
renderer.render(scene, camera);
cssRenderer.render(cssScene, camera);
});

start();
});
```

## 4 Відстеження обличчя

### 4.1 Опорні точки обличчя

Бібліотека MindAR має два основні набори модулів – для роботи з зображеннями (`image`) та для роботи з обличчями (`face`), – що можуть використовуватись одночасно, але потребують окремого підключення у документі HTML:

```
<html>
  <head>
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0">
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-face-three.prod.js"
      ></script>
    <script type="importmap">
    {
      "imports": {
        "three": "https://unpkg.com/three/build/three.module.js"
      }
    }
    <script src=".main.js"></script>
    <style>
      html, body {position: relative; margin: 0;
                  width: 100%; height: 100%; overflow: hidden}
    </style>
  </head>
  <body>
  </body>
</html>
```

Схожість API для відстеження зображень і відстеження обличчі простежується у коді `main.js`. Так, спочатку у аналогічний спосіб відбувається створення посилання на об'єкти Three.js, але не з `window.MINDAR.IMAGE.THREE`, а з `window.MINDAR.FACE.THREE`:

```
const THREE = window.MINDAR.FACE.THREE;
```

Так само всі дії виконуються у функції `start`, що виключається після завантаження документу HTML. Відмінність полягає у застосуванні конструктора класу `MindARThree` з `window.MINDAR.FACE.THREE` замість `window.MINDAR.IMAGE`. Через те, що бібліотека розпізнає обличчя, параметр `imageTargetSrc` для цього конструктора не потрібен:

```
document.addEventListener('DOMContentLoaded', () => {
  const start = async() => {
    const mindarThree = new window.MINDAR.FACE.MindARThree({
      container: document.body,
    });
  };
});
```

Для створення об'єктів Three.js застосовуються властивості об'єкта `mindarThree`: рендерер, сцена, камера:

```
const {renderer, scene, camera} = mindarThree;
```

Створимо якір та прив'яжемо до нього напівпрозору сферу:

```
const geometry = new THREE.SphereGeometry(0.1, 32, 16);
const material = new THREE.MeshBasicMaterial({color: 0x00ffff,
  transparent: true, opacity: 0.5});
const sphere = new THREE.Mesh(geometry, material);

const anchor = mindarThree.addAnchor(1);
anchor.group.add(sphere);

await mindarThree.start();
renderer.setAnimationLoop(() => {
  renderer.render(scene, camera);
});
start();
});
```

Попри схожість, метод `addAnchor` по-іншому трактує параметр: якщо для `window.MINDAR.IMAGE` це був номер цільового зображення, то при розпізнаванні облич ще буде номер опорної точки обличчя. Точка 1 – це ніс (рис. 4.1).



Рис. 4.1. Прив'язування об'єкту до опорної точки обличчя.

Виявлення опорних точок обличчя базується на відомій моделі бібліотеки TensorFlow (<https://github.com/tensorflow/tfjs-models/tree/master/face-landmarks-detection>).

Модель MediaPipe Face Mesh (<https://google.github.io/mediapipe/solutions/models.html>) є згортковою нейронною мережею, що визначає на обличчі 468 тривимірних опорних точок ([https://github.com/tensorflow/tfjs-models/raw/master/face-landmarks-detection/mesh\\_map.jpg](https://github.com/tensorflow/tfjs-models/raw/master/face-landmarks-detection/mesh_map.jpg)), і ми можемо прив'язувати об'єкти до будь-якої з них (рис. 4.2).

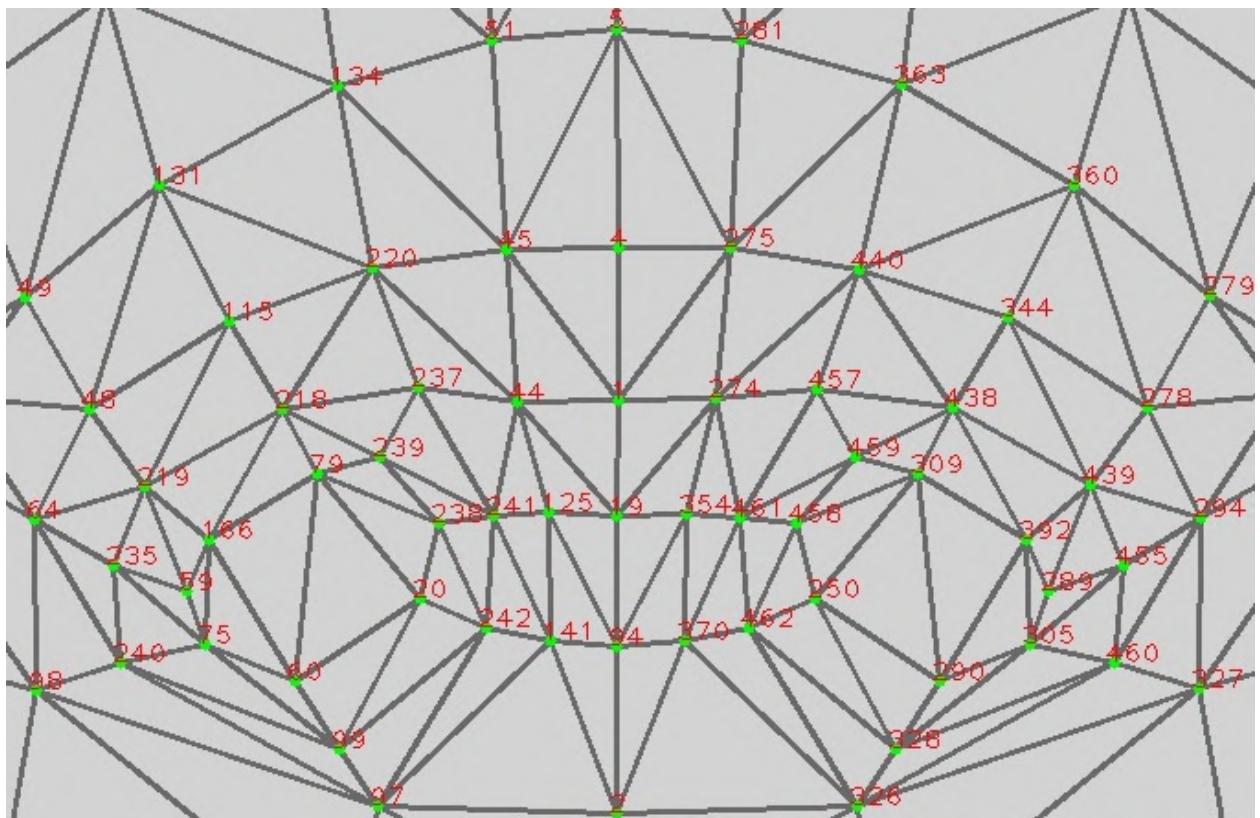


Рис. 4.2. Опорні точки обличчя (фрагмент).

## 4.2 Проблема оклюзії

Спробуємо надіти обличчя окуляри, для чого застосуємо їх довільну модель у форматі GLTF (наприклад, <https://www.cgtrader.com/free-3d-print-models/fashion/glasses/3d-oval-glasses-3d-print-model>). Для цього скористаємось раніше створеною бібліотекою завантажувачів loader.js:

```
import {loadGLTF} from "./loader.js";
```

Для освітлення моделі застосуємо напівсферичне джерело світла:

```
const light = new THREE.HemisphereLight( 0xffffff, 0xbbbbff, 1 );
scene.add(light);
```

Моделі у форматі GLTF можуть зберігатися як у вигляді множині файлів опису та текстур, так й одному бінарному файлі із розширенням .glb

```
const glasses = await loadGLTF("glasses.glb");
```

Масштабування моделі виконано під розмір обличчя:

```
glasses.scene.scale.set(0.22, 0.22, 0.22);
```

Модель прив'язано до перенісся (точка 168 між очима, рис. 4.3):

```
const anchor = mindarThree.addAnchor(168);  
anchor.group.add(glasses.scene);
```

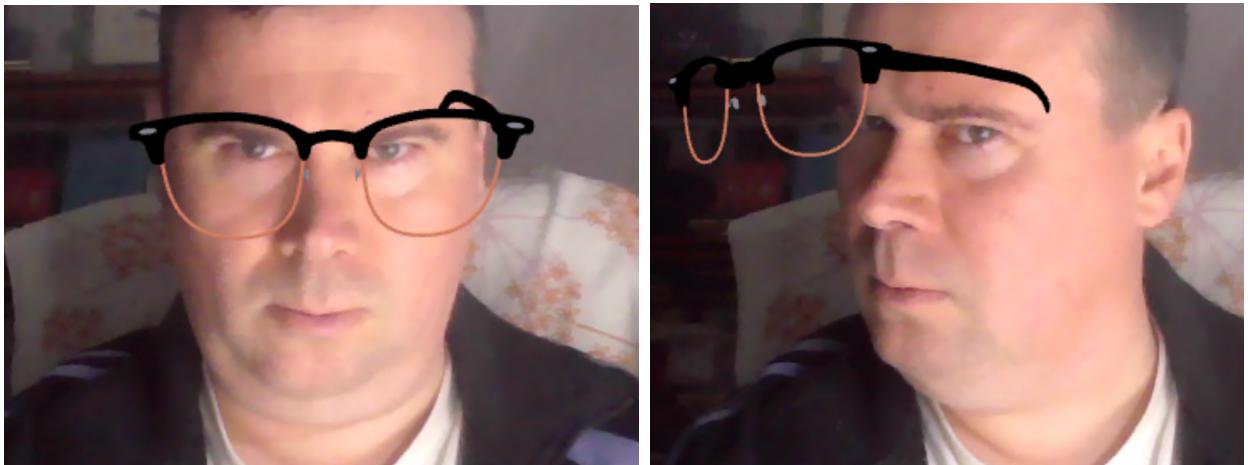


Рис. 4.3. Модель окулярів, прив'язана до перенісся.

Праворуч на рис. 4.3 продемонстровано, що дужка окулярів знаходить не на місці, а самі окуляри розташовані досить далеко від обличчя. Намагання їх змістити за віссю  $z$  призведе до корекції віддалення від обличчя, проте ефект того, що дужка рухається по ньому, залишається:

```
glasses.scene.position.set(0, -0.12, -0.70);
```

*Оклюзія* – ситуація, в якій два об'єкти розташовані приблизно на одній лінії і один об'єкт, розташований близче до віртуальної камери або вікна перегляду, частково або повністю закриває видимість іншого об'єкта. Для подолання цієї проблеми у графічному конвеєрі використовується “оклюзивне обрізання” для видалення прихованих поверхонь перш, ніж до них почнуть застосовуватися растеризація і шейдери.

Доповнена реальність створюється накладанням полотна поверх відео з веб-камери. Це означає, що все, що рендериться на верхньому полотні, буде приховувати нижній шар відео, тому, хоча інтуїтивно голова мала б приховати дужку окулярів, цього не відбулося.

Ідея оклюзивного обрізання полягає в тому, що на 3D сцені створюються об'єкти, які нагадують фізичні об'єкти, які ми хочемо використовувати, щоб приховати що-небудь за ними – оклюдери.

У даному прикладі фізичним об'єктом є голова, тому оклюдером має бути віртуальна голова на полотні: тієї ж форми, що й реальна, але при цьому невидима. Незважаючи на невидимість, вона повинна, однак, ще мати можливість приховувати об'єкти, які знаходяться за нею.

У документації до Meta Spark Studio <https://sparkar.facebook.com/ar-studio/learn/articles/people-tracking/face-reference-assets> є посилання на модель голови у форматах DAE та OBJ – останній можна конвертувати у формат GLTF, наприклад, онлайн-конвертором (<https://products.apose.app/3d/conversion/obj-to-glb>). У форматі GLTF можна знайти іншу модель голови від 8th Walls (<https://8thwall.8thwall.app/face-effects-physics/assets/head-occluder-1-8ns1sijd87.glb>).

Спочатку необхідно завантажити модель-оклюдер та емпірично налаштувати її розміри та положення:

```
const occluder = await loadGLTF("headOccluder.glb");

occluder.scene.scale.set(0.065, 0.065, 0.065);
occluder.scene.position.set(0, -0.3, 0.15);
```

Наступним кроком є забезпечення “правильної” прозорості оклюдера – якщо просто встановити для неї параметри матеріалу `transparent` у `true` та `opacity` у 1.0, окуляри просвічуватимуться, а не відсікатися. Для досягнення необхідного ефекту створимо матеріал із властивістю `colorWrite`, яка визначає, чи рендерити колір матеріалу. Це може бути використано разом з властивістю `renderOrder` для створення невидимого об'єкту, які закривають інші об'єкти:

```
const occluderMaterial =
  new THREE.MeshBasicMaterial({colorWrite: false});

occluder.scene.renderOrder = 0;
```

Властивість `renderOrder` визначає порядок рендерингу. Для оклюдера вона встановлена у 0 (рендерити першим), а для окулярів – у 1 (рендерити другим):

```
glasses.scene.renderOrder = 1;
```

Метод `traverse` застосовується для того, щоб викликати функцію для об'єкта та всіх його нащадків. Ураховуючи, що модель є ієархією об'єктів, застосуємо його для того, щоб у тих з них, які є об'єктними сітками, замінити матеріал на невидимий:

```
occluder.scene.traverse((o) => {
  if (o.isMesh) {
    o.material = occluderMaterial;
  }
});
```

Останнім кроком є створення якоря для оклюдера, який також, як і якір для окулярів, прив'язаний до переносиці:

```
const occluderAnchor = mindarThree.addAnchor(168);
occluderAnchor.group.add(occluder.scene);
```

Результат цілком працездатний, хоча й далекий від ідеального (рис. 4.4).

Причина в тому, що оклюдер має дещо іншу форму, ніж голова на рис. 4.4. Крім того, положення голови оцінюється, використовуючи опорні точки на обличчі, що визначаються неточно.

Позиціювання буде працювати краще, якби голова була поголена, а обличчя б не мало суттєвих жирових відкладень. Альтернативним варіантом забезпечення точності є створення точних моделей конкретних голів.

## 4.3 Накладання маски на обличчя

Маска для обличчя (face mesh) – ще один тип дополненої реальності, пов'язаної з накладання зображень (текстур) на всі опорні точки обличчям



Рис. 4.4. Демонстрація роботи оклюдера.

людини, а не прив'язки до окремих із них. Маски для обличчя використовуються для створення різноманітних ефектів макіяжу, татуювання тощо – аж до повної віртуалізації особи.

Маска для обличчя не є заздалегідь визначеною 3D-моделлю – вона динамічно генерується з постійним оновленням геометрії.

Для накладання маски на обличчя нам знадобиться відповідна текстура, для завантаження якої застосовується метод `loadTexture` з користувальської бібліотеки `loader.js`:

```
import {loadTexture} from "./loader.js";
```

Створення маски відбувається викликом `addFaceMesh`:

```
const faceMesh = mindarThree.addFaceMesh();
```

Метод `addFaceMesh` за формою схожий на `addAnchor`, але сутність у них різна: у `addAnchor` створюється порожня група, до якої додаються

об'єкти, положенням яких керує MindAR, у той час як `faceMesh`, що повертається `addFaceMesh`, є єдиним відображуваним об'єктом, геометрія якого змінюється у кожному кадрі.

Матеріалом маски для обличчя може бути довільна текстура – якщо її не встановити, маска для обличчя буде виглядати, як показано на першому зображені (рис. 4.5).

Побачити структуру цієї маски можна на другому зображені (рис. 4.5) – для цього необхідно встановити атрибут `wireframe` матеріалу зображення:

```
faceMesh.material.wireframe = true;
```

Трете, четверте та п'яте зображення (рис. 4.5) є прикладами накладання модифікованих та оригінальної ([https://github.com/google/mediapipe/blob/master/mediapipe/modules/face\\_geometry/data/canonical\\_face\\_model\\_uv\\_visualization.png](https://github.com/google/mediapipe/blob/master/mediapipe/modules/face_geometry/data/canonical_face_model_uv_visualization.png)) текстур опорних точок обличчя. У документації до Meta Spark Studio (<https://sparkar.facebook.com/ar-studio/learn/articles/people-tracking/face-reference-assets#optimization>) можна знайти набір текстур для масок для обличчя, що можна використовувати для створення власних масок, як описано у <https://sparkar.facebook.com/ar-studio/learn/articles/creating-and-prepping-assets/the-face-mask-template-in-Adobe/>.

В якості прикладу застосуємо текстуру з <https://tinyurl.com/3nx53bek> (рис. 4.5, шосте зображення):

```
const texture = await loadTexture("moustache1.png");
faceMesh.material.map = texture;
```

Текстура `texture`, встановлена в якості матеріалу маски для обличчя, містить певні прозорі елементи, для підтримки яких встановлюється властивість матеріалу `transparent` (рівень прозорості регулюється значенням `opacity`):

```
faceMesh.material.transparent = true;
```

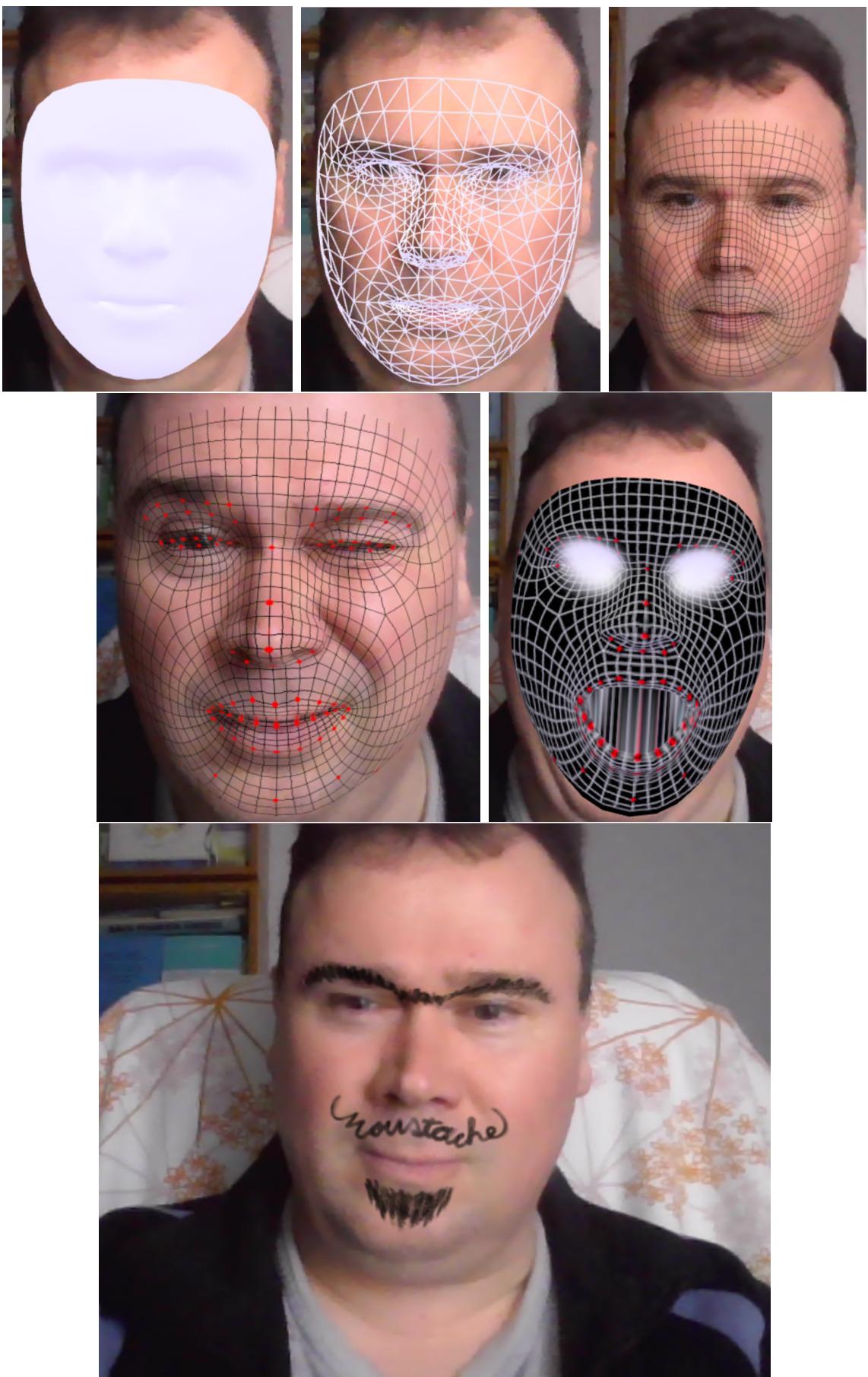


Рис. 4.5. Маски для обличчя.

Встановлення параметру `needsUpdate` необхідно для вказання на необхідність оновлення текстури при зміні геометрії:

```
faceMesh.material.needsUpdate=true;
```

Останній крок – розміщення маски на сцені; в ідеалі, маска для обличчя повинна мати таку ж геометрію та положення, як і саме обличчя:

```
scene.add(faceMesh);
```

Створення гарної маски потребує певних художніх навичок, проте, послуговуючись канонічною текстурою (рис. 4.2), це зробити досить просто – достатньо нанести поверх неї необхідне зображення та видалити зайді лінії.

## 4.4 Перемикання камери

Сьогодні більшість мобільних пристройв має дві камери, передню (фронтальну) та задню (тильну), тому у WebAR додатках можливість перемикання між ними – важливий складник функціональності (рис. 4.6).

Для її реалізації у документі HTML доцільно створити кнопку:

```
<button id="switch">Переключитись на іншу камеру</button>
```

Розміщення кнопки у документі визначається стилювим описом:

```
<style>
#switch {
    position: fixed;
    bottom: 5px;
    left: 50%;
    transform: translateX(-50%);
    z-index: 10;
}
</style>
```

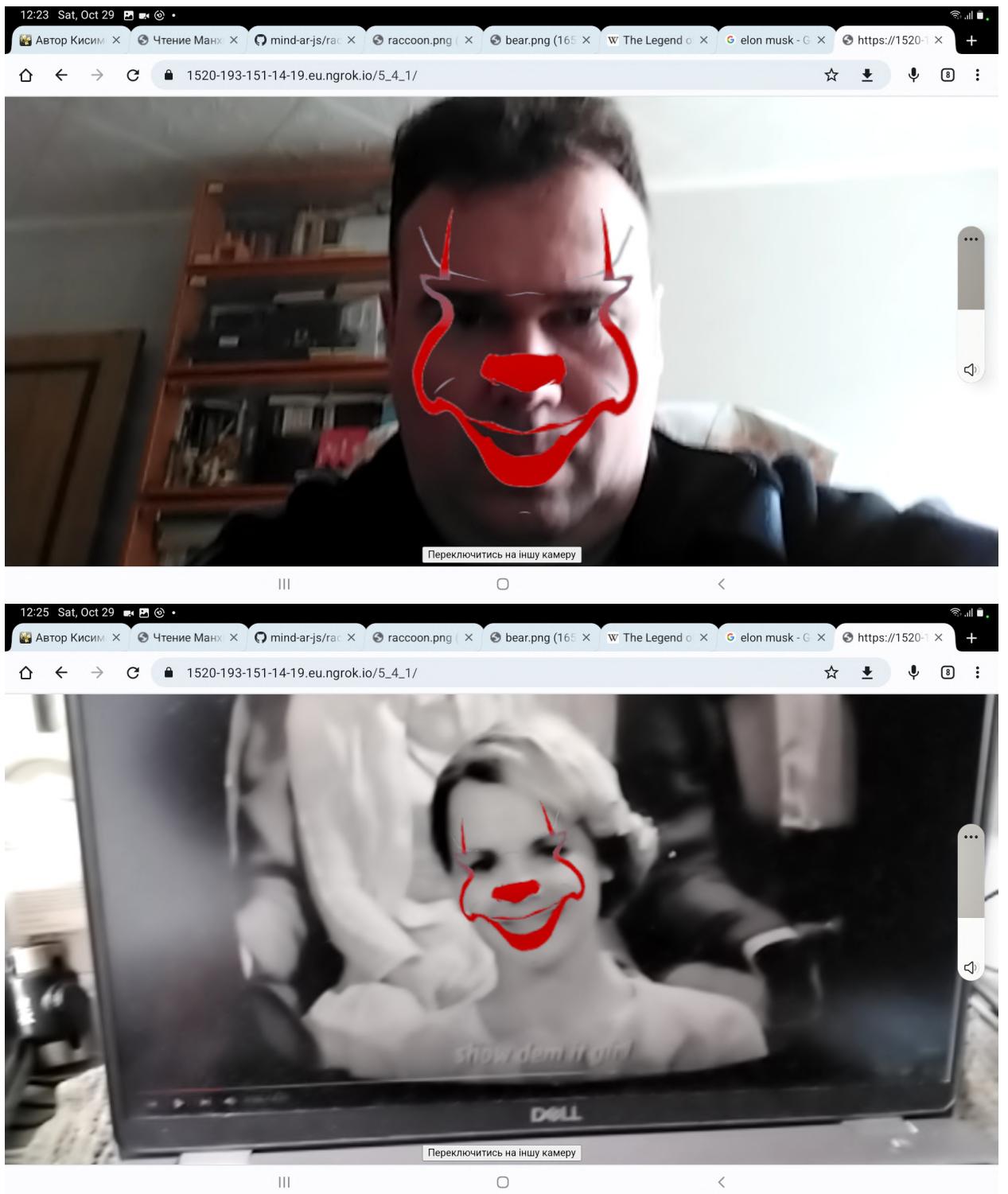


Рис. 4.6. Доповнена реальність на зображеннях із фронтальної та тильної камер.

Встановлення `position: y: fixed` означає, що для кнопки в макеті сторінки не створюється окреме місце – вона позиціонується відносно блоку, встановленого `viewport`. Її кінцева позиція визначається значеннями `top`, `right`, `bottom` і `left` – останні два встановлені явно.

Властивість `z-index` встановлена у досить велике значення, що визначає розташування кнопки поверх елементів із меншим, ніж 10, значенням `z-index`.

Властивість `transform` надає можливість обертати, масштабувати, нахиляти та зміщувати кнопку – останнє реалізовано викликом `translateX` вліво на половину ширини `viewport`.

По натисканню кнопки викликається метод переключення на іншу камеру:

```
document.querySelector("#switch").addEventListener("click", () => {
    mindarThree.switchCamera();
});
```

## 4.5 Захоплення кадрів

Інша бажана функціональність – захоплення кадрів: створення фотознімку сцени у доповненій реальності. MindAR не надає для неї певного стандартного методу, проте створення власної функції для захоплення кадрів не є проблематичним. При цьому можна не обмежуватись лише кадром – за потреби, до фото можна додати рамку, логотип, підпис тощо.

Для її реалізації у документі HTML доцільно створити кнопку – цього разу без надпису:

```
<button id="capture"></button>
```

Надпис не потрібен тому, що кнопка однозначно ідентифікується формою – значення властивості стилю `border-radius` від 50% до 100% роблять її круглою:

```
<style>
#capture {
    position: fixed;
    bottom: 2vh;
    left: 50%;
    height: 12vw;
    width: 12vw;
```

```
border: solid 2px;  
border-radius: 50%;  
transform: translateX(-50%);  
z-index: 10;  
}  
</style>
```

По натисканню кнопки викликається метод захоплення кадру відео – даний фрагмент коду необхідно додати до функції `start`. Параметром методу `capture` є об'єкт AR-рушія `mindarThree`:

```
document.querySelector("#capture").addEventListener("click", () => {  
    capture(mindarThree);  
});
```

Визначити його можна за межами обробника події `DOMContentLoaded`:

```
const capture = (mindarThree) => {  
}
```

З об'єкту `mindarThree` отримаємо те, що необхідно для виконання знімку – відеопотік, рендерер, сцену та камеру:

```
const {video, renderer, scene, camera} = mindarThree;
```

Для створення доповненого фото нам необхідно спочатку взяти кадр з відеопотоку `video`, а потім на нього накласти зображення з полотна `renderCanvas`, яке є складовою документа, пов'язаною з рендерером:

```
const renderCanvas = renderer.domElement;
```

Для суміщення цих зображень нам знадобиться буфер – ще одне полотно `canvas`, яке існуватиме виключно у пам'яті й не буде візуалізуватись:

```
const canvas = document.createElement('canvas');
```

Фото є двовимірним об'єктом, тому для отримання контексту малювання викликаємо метод `getContext` із параметром `2d`:

```
const context = canvas.getContext("2d");
```

Цей метод, як й ті, що використовуватимуться надалі, не є методами Three.js – вони належать до низькорівневого Web API, а саме до інтерфейсу HTMLCanvasElement (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement>). Звернення до нього зумовлене тим, що відеопотік з камери як такий не контролюється бібліотекою Three.js.

Розміри полотна-буфера для доповненого фото встановимо у розміри полотна, на якому відображені об'єкти Three.js:

```
canvas.width = renderCanvas.width;  
canvas.height = renderCanvas.height;
```

За бажання, до полотна-буфера можна додати рамку – тоді розміри доведеться збільшити на її подвоєні ширину та висоту. Полотно-буфер може бути й меншого розміру, але тоді при копіюванні відеокадру та проекції 3D-сцени їх доведеться пропорційно масштабувати.

Всі дії над вмістом буферного полотна виконуються через його контекст малювання. Метод `drawImage` виконує копіювання на буферне полотно `canvas` зображення, що є першим параметром методу – у нашому випадку одного кадру відеопотоку `video`:

```
const sx = (video.clientWidth - renderCanvas.clientWidth) / 2  
    * video.videoWidth / video.clientWidth;  
const sy = (video.clientHeight - renderCanvas.clientHeight) / 2  
    * video.videoHeight / video.clientHeight;  
const sw = video.videoWidth - sx * 2;  
const sh = video.videoHeight - sy * 2;  
  
context.drawImage(video, sx, sy, sw, sh,  
    0, 0, canvas.width, canvas.height);
```

Інші параметри `drawImage` – координати (`sx, sy`) верхнього лівого кута зображення, що копіюється, його ширина `sw` та висота `sh`, а також аналогічний набір для місця, куди виконується копіювання: `(0, 0)` відповідає

початку буферного полотна, `canvas.width` та `canvas.height` – його ширині та висоті.

Обчислення `sx`, `sy`, `sw`, `sh` необхідно як тому, що розміри відео та полотна не співпадають (рис. 4.7), так й тому, що полотно може покривати не все відео (наприклад, через ненульові границі відео може симетрично виходити за межі полотна).

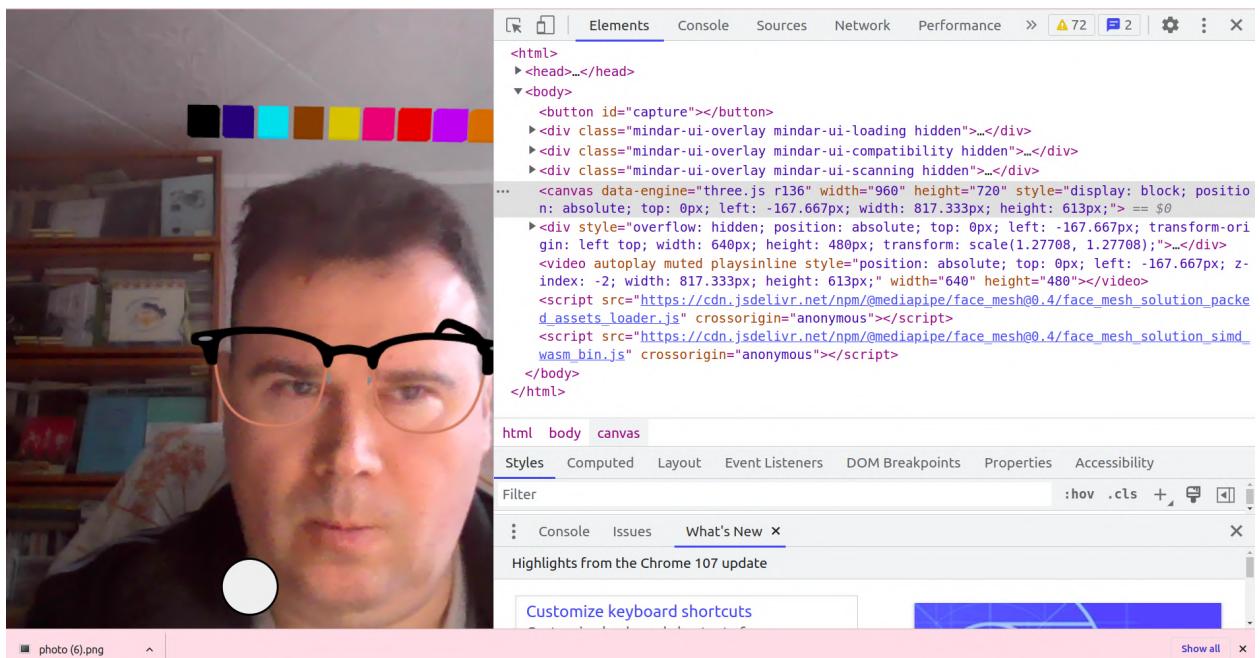


Рис. 4.7. Приклад захоплення доповненого фото.

Скопіювати до буферного полотна зображення з полотна `renderCanvas`, що містить об'єкти доповненої реальності, суттєво простіше, адже вони однакового розміру, тому при виклику `drawImage` вказується лише один набір параметрів – другий вважається йому ідентичним. Проте одного виклику `drawImage` недостатньо: цілком може статись, що скопіюється порожнє зображення через те, що на кожному кадрі полотно `renderCanvas` очищується й усі об'єкти на ньому відображаються заново – ми просто можемо потрапити на момент очищення. Щоб цього запобігти, необхідно встановити значення `preserveDrawingBuffer` у `true` для того, щоб зберігати буфери до ручного очищення або перезапису, після чого явно виконати перезапис буфера актуальним зображенням викликом `render`:

```
renderer.preserveDrawingBuffer = true;  
renderer.render(scene, camera);
```

```
context.drawImage(renderCanvas, 0, 0, canvas.width, canvas.height);
renderer.preserveDrawingBuffer = false;
```

Отже, у буферному полотні збережені як кадр з відео, масштабований до розмірів полотна з об'єктами Three.js, так й саме це полотно із об'єктами Three.js – якщо присутні об'єкти доповненої реальності, реалізовані через CSS, необхідно отримати ще одно полотно для них із рендереру CSS та у аналогічний спосіб скопіювати його вміст до буферного полотна (пропонуємо це як вправу).

Метод `toDataURL` повертає URL-адресу даних `data`, що містить подання зображення роздільною здатністю 96 dpi у форматі, заданому його параметром – якщо формат файлу не вказано, або якщо даний формат не підтримується, то дані будуть експортовані у форматі `image/png`:

```
const data = canvas.toDataURL("image/png");
```

Повернути ці дані користувачу можна, створивши тимчасове безіменне гіперпосилання, яке, так само як і буферне полотно, не додається до документу, а існує лише в пам'яті. Метод `click` імітує натискання на гіперпосилання – у результаті користувач отримає файл з ім'ям, заданим у властивості `download`, та вмістом `data`, заданим властивістю `href`:

```
const link = document.createElement("a");
link.download = "photo.png";
link.href = data;
link.click();
```

## 4.6 Застосування Web Share API на мобільних пристроях

Продовжуючи приклад із захопленням доповнених фотографій, розглянемо, як поділитися захопленими (та, звичайно, захоплюючими) кадрами у соціальних мережах за допомогою `Web Share API`. Для цього, крім кнопки `capture` для захоплення фото із попереднього прикладу, передбачимо блок попереднього перегляду, до складу якого включимо три елементи – `preview-image` (зображення для попереднього перегляду), `preview-close`

(контейнер із надписом Х, що виконуватиме роль кнопки) та preview-share (аналогічний контейнер із надписом Поділитись):

```
<button id="capture"></button>
<div id="preview">
  <div id="preview-close">X</div>
  <img id="preview-image"/>
  <div id="preview-share">Поділитись</div>
</div>
```

До блоку опису стилів додамо описи нових елементів:

- блок попереднього перегляду – прихований прямокутник посередині головного вікна документу, що має 80% його розміру та обрамлений білою рамкою товщиною 10 пікселів:

```
#preview { position: fixed; z-index: 10; left: 10%; top: 10%;
            width: 80%; height: 80%; border: solid 10px white;
            visibility: hidden; }
```

- зображення для попереднього перегляду займатиме 100% розміру блоку, в який воно вкладене:

```
#preview-image { width: 100%; height: 100%; }
```

- “кнопка” закриття розташована поверх зображення у його правому верхньому куті:

```
#preview-close { position: absolute; font-size: 20px;
                  padding: 5px; border: solid 1px;
                  cursor: pointer; color: white;
                  border: solid 5px white;
                  right: -5px; top: -5px; }
```

- блок елементів, за допомогою яких можна буде поділитись зображенням, розташований у нижній частині на півширини основного вікна:

```
#preview-share { border: solid 1px; text-align: center;
    cursor: pointer; background: white;
    position: absolute; left: 50%;
    transform: translateX(-50%); bottom: 10px;
    padding: 10px 30px; border-radius: 10px;
    border: none; }
```

Приберемо з функції `capture` блок завантаження файлу – тепер вона буде повертати саме захоплене зображення:

```
const capture = (mindarThree) => {
    const {video, renderer, scene, camera} = mindarThree;
    const renderCanvas = renderer.domElement;
    const canvas = document.createElement("canvas");
    const context = canvas.getContext("2d");
    canvas.width = renderCanvas.width;
    canvas.height = renderCanvas.height;
    const sx = (video.clientWidth - renderCanvas.clientWidth) / 2
        * video.videoWidth / video.clientWidth;
    const sy = (video.clientHeight - renderCanvas.clientHeight) / 2
        * video.videoHeight / video.clientHeight;
    const sw = video.videoWidth - sx * 2;
    const sh = video.videoHeight - sy * 2;
    context.drawImage(video, sx, sy, sw, sh,
        0, 0, canvas.width, canvas.height);
    renderer.preserveDrawingBuffer = true;
    renderer.render(scene, camera);
    context.drawImage(renderCanvas, 0, 0, canvas.width, canvas.height);
    renderer.preserveDrawingBuffer = false;
    const data = canvas.toDataURL("image/png");
    return data;
}
```

Основні зміни – у функції `start`. Спочатку створимо об'єкти для маніпулюванні із контейнерами, вкладеними у `preview`. Для цього використаємо метод `document.querySelector` з їх ідентифікаторами:

```
const previewImage = document.querySelector("#preview-image");
const previewClose = document.querySelector("#preview-close");
const preview = document.querySelector("#preview");
const previewShare = document.querySelector("#preview-share");
```

Далі змінимо обробник натискання на кнопку `capture` – отримавши захоплене зображення, передамо його до елементу `preview-image` документу HTML та зробимо його видимим:

```
document.querySelector("#capture").addEventListener("click", () => {
  const data = capture(mindarThree);
  preview.style.visibility = "visible";
  previewImage.src = data;
});
```

У результаті цієї дії захоплене зображення з'явиться у “вікні” попереднього перегляду та буде масштабовано під його розмір без дотримання пропорцій (рис. 4.8).

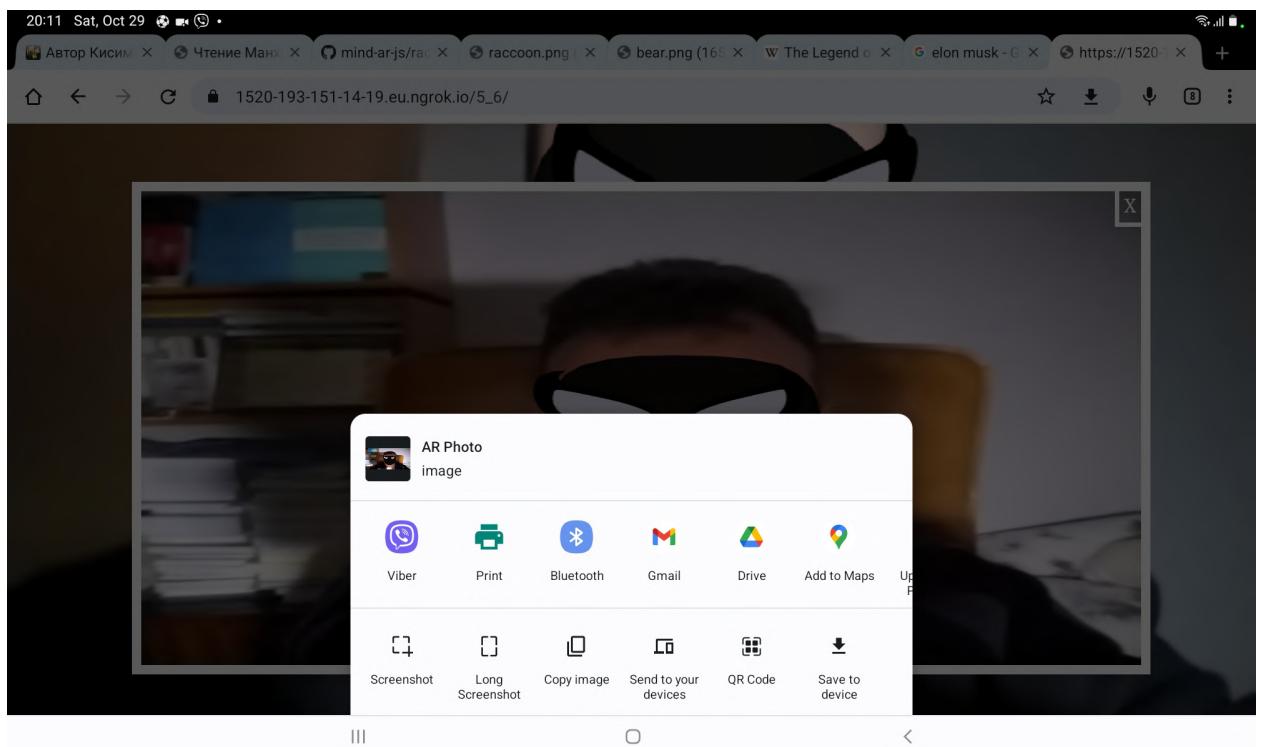


Рис. 4.8. Приклад роботи Web Share API на мобільному пристрої.

При натисканні в області попереднього перегляду відповідний елемент приховується:

```
previewClose.addEventListener("click", () => {
    preview.style.visibility = "hidden";
});
```

При натисканні на Поділитись створюємо буферне полотно `canvas`, до якого копіюємо захоплене та збережене у `previewImage` зображення:

```
previewShare.addEventListener("click", () => {
    const canvas = document.createElement("canvas");
    canvas.width = previewImage.width;
    canvas.height = previewImage.height;
    const context = canvas.getContext("2d");
    context.drawImage(previewImage, 0, 0, canvas.width, canvas.height);
```

Метод `toBlob` створює об'єкт `blob` класу `Blob` – зображення із має роздільною здатністю 96 дрі, яке міститься в `canvas`. Параметром `toBlob` є функція зворотного виклику:

```
canvas.toBlob((blob) => {
    const file = new File([blob], "photo.png", {type: "image/png"});
    const files = [file];
    if (navigator.canShare && navigator.canShare({files})) {
        navigator.share({files: files, title: "AR Photo", })
    } else {
        const link = document.createElement("a");
        link.download = "photo.png";
        link.href = previewImage.src;
        link.click();
    }
});
```

Нащадком класу `Blob` є клас `File`, конструктор якого приймає масив даних про зображення `blob`, ім'я та MIME-тип файлу. Web Share API передбачає, що поділитись можна множиною файлів, тому створюється масив `files`, до якого включається один єдиний елемент – сконструйований об'єкт `file`.

Доступність Web Share API поки що обмежена, тому перед його використанням слід перевірити, чи він підтримується, викликами `navigator.canShare`: перший виклик (без параметрів) перевіряє доступність Web Share API, а другий – можливість поділитись файлами з масиву `files`: метод `navigator.canShare` повертає `true`, якщо еквівалентний виклик `navigator.share` був би успішним, та `false`, якщо дані не можуть бути перевірені. Через те, що Web Share API обмежений політикою дозволу на спільний доступ до веб-ресурсу, метод `navigator.canShare` поверне `false`, якщо дозвіл підтримується, але не був наданий.

За успішної перевірки викликається метод `navigator.share`, аргументами якого є масив `files` об'єктів класу `File` та необов'язковий заголовок `title`. За бажанням, можна поділитись також посиланням `url` та рядком `text`.

Якщо файлами не можна поділитись за допомогою Web Share API, використовуємо спосіб із попереднього прикладу – автозавантаження зображення на пристрій.

Існують альтернативи застосуванню Web Share API – так, можна було б налаштувати сервер, на який завантажувати захоплене зображення і генерувати публічну URL-адресу.

Інший варіант – спробувати натиснути на зображення для попереднього перегляду правою кнопкою миші та зберегти його або натиснути пальцем у мобільному браузері й утримувати це зображення – з'являється спливаюче вікно, що дозволяє легко поділитися зображенням. Тоді для користувача доцільно було б вивести інструкцію типу “Натисніть і утримуйте зображення для того, щоб поділитись ним”.

## 4.7 Приклад: примірка віртуальних аксесуарів

Віртуальна примірка<sup>2</sup> є одним із доцільних застосувань доповненої реальності, пов'язаної з відстеженням обличчя. Hiukim Юен пропонує приклад такої примірочної (<https://hiukim.github.io/mind-ar-js-doc/face-tracking-examples/tryon>), у якій користувачі можуть вибирати і приміряти різні віртуальні аксесуари – скористаємося ним для того, щоб

<sup>2</sup>Див., наприклад, *Examples of AR-powered virtual try ons in the fashion industry* (<https://www.divante.com/blog/examples-of-ar-powered-virtual-try-ons-in-the-fashion-industry>)

показати, як може працювати такий додаток.

Для створення власної примірочної необхідно, по-перше, завантажити різні аксесуари, в ідеалі – однотипні, але можуть бути й різних типів, кожен із яких потребує власну точку для прикріплення на обличчі.

У прикладі пропонуються три типи аксесуарів – капелюхи, окуляри та сережки, які можна замінити та доповнити на інші безкоштовні моделі, знайдені в Інтернет. Початкове налаштування їх розмірів та позиції виконується емпірично.

По-друге, необхідно створити певний інтерфейс для управління вибором. Так, можна просто показати всі аксесуари, якщо їх небагато, але за великої кількості доцільно надати можливість вибору спочатку типу аксесуару, а потім вже конкретного екземпляру.

По-третє, необхідно надати можливість ділитись результатами примірки через Web Share API (рис. 4.9).

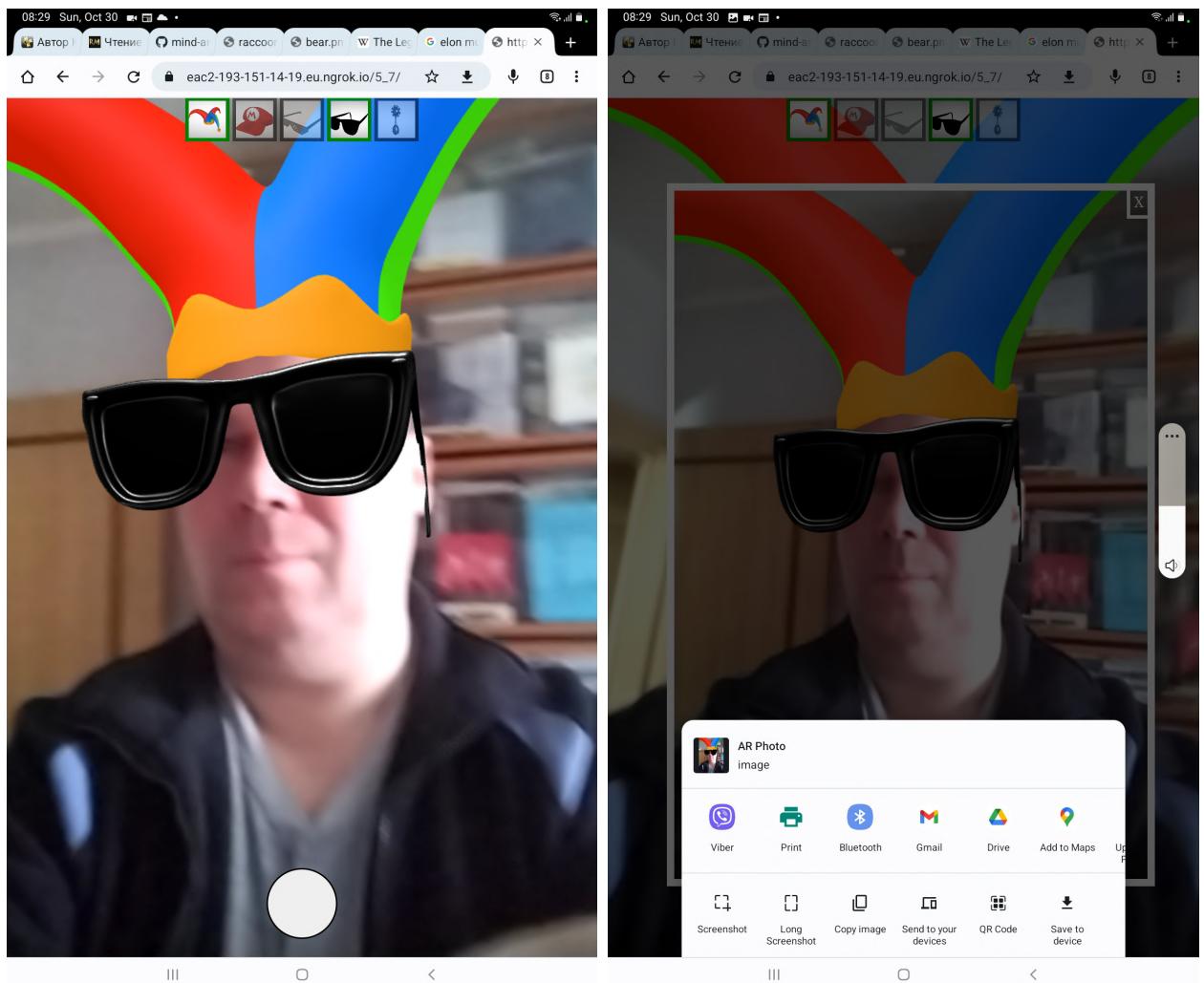


Рис. 4.9. Примірка віртуальних аксесуарів.

Серед інших ідей, що можуть бути реалізовані з використанням масок для обличчя – вибір типу макіяжу, татуювання тощо.

Початок документу HTML віртуальної примірочної типовий – налаштування області відображення, підключення MindAR та Three.js є основної програми:

```
<html>
  <head>
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-face-three.prod.js"
     ></script>
    <script type="importmap">
    {
      "imports": {
        "three": "https://unpkg.com/three/build/three.module.js"
      }
    }
    </script>
    <script src="./main.js" type="module"></script>
```

Типовим є й налаштування CSS для документа та його тіла:

```
<style>
  html, body { position: relative; margin: 0; width: 100%;
                height: 100%; overflow: hidden }
```

На рис. 4.9 (ліворуч) показані елементи блоку вибору аксесуарів **selections**, розташовані у верхній частині по центру:

```
#selections { position: fixed; left: 50%; top: 0; z-index: 2;
               width: 100%; text-align: center;
               transform: translateX(-50%); }
```

У блоці **selections** розміщаються напівпрозорі зображення, масштабовані до розміру  $50 \times 50$  зі збереженням пропорцій, обмежені рамкою товщиною 5 пікселів:

```
#selections img { border: solid 5px; width: 50px; height: 50px;  
    object-fit: cover; cursor: pointer;  
    opacity: 0.5; }
```

Коли аксесуар обирається, його зображення стає непрозорим, а рамка – зеленого кольору:

```
#selections img.selected { border-color: green; opacity: 1; }
```

Стильовий опис блоку `preview` елементів для попереднього перегляду та поширення доповненого фото (`preview-image`, `preview-close` та `preview-share`) та кнопки для його захоплення `capture` такий самий, як і у попередньому прикладі:

```
#preview { position: fixed; z-index: 10; left: 10%; top: 10%;  
    width: 80%; height: 80%; border: solid 10px white;  
    visibility: hidden; }  
  
#preview-image { width: 100%; height: 100%; }  
  
#preview-close { position: absolute; right: 0; top: 0;  
    font-size: 20px; padding: 5px;  
    border: solid 1px; cursor: pointer;  
    color: white; border: solid 5px white;  
    right: -5px; top: -5px; }  
  
#preview-share { border: solid 1px; text-align: center;  
    cursor: pointer; background: white;  
    position: absolute; left: 50%;  
    transform: translateX(-50%); bottom: 10px;  
    padding: 10px 30px; border-radius: 10px;  
    border: none; }  
  
#capture { position: fixed; bottom: 2vh; left: 50%;  
    height: 12vw; width: 12vw; border: solid 2px;  
    border-radius: 50%; transform: translateX(-50%);  
    z-index: 10; }  
  
</style>  
</head>
```

Тіло документа складається з контейнеру **selections**, кнопки **capture** та контейнеру **preview**. До складу контейнеру **selections** входять 5 зображень попереднього перегляду аксесуарів із ідентифікаторами **hat1**, **hat2**, **glasses1**, **glasses2** та **earring** – останньому відповідатимуть два об'єкти (сережки на ліве та праве вухо):

```
<body>
  <div id="selections">
    
    
    
    
    
  </div>
```

Далі створюється кнопка **capture** та контейнер **preview**, який, у свою чергу, містить контейнер **preview-close**, зображення **preview-image** та контейнер **preview-share**:

```
<button id="capture"></button>
<div id="preview">
  <div id="preview-close">X</div>
  <img id="preview-image"/>
  <div id="preview-share">Share</div>
</div>
</body>
</html>
```

`main.js` розпочинається з імпорту завантажувача моделей `loadGLTF` (саме для нього в опису документа був визначений символ `three` у карті імпорту) та отримання посилання `THREE` на бібліотеку `Three.js`:

```
import {loadGLTF} from "./loader.js";
const THREE = window.MINDAR.FACE.THREE;
```

Функція `capture` узята із попереднього прикладу без змін:

```
const capture = (mindarThree) => {
    const {video, renderer, scene, camera} = mindarThree;
    const renderCanvas = renderer.domElement;
    const canvas = document.createElement("canvas");
    const context = canvas.getContext("2d");
    canvas.width = renderCanvas.clientWidth;
    canvas.height = renderCanvas.clientHeight;
    const sx = (video.clientWidth - renderCanvas.clientWidth) / 2
               * video.videoWidth / video.clientWidth;
    const sy = (video.clientHeight - renderCanvas.clientHeight) / 2
               * video.videoHeight / video.clientHeight;
    const sw = video.videoWidth - sx * 2;
    const sh = video.videoHeight - sy * 2;
    context.drawImage(video, sx, sy, sw, sh,
                     0, 0, canvas.width, canvas.height);
    renderer.preserveDrawingBuffer = true;
    renderer.render(scene, camera);
    context.drawImage(renderCanvas, 0, 0, canvas.width, canvas.height);
    renderer.preserveDrawingBuffer = false;
    const data = canvas.toDataURL("image/png");
    return data;
}
```

Після завантаження документу HTML спрацьовує обробник події `DOMContentLoaded`, за якою визначається та викликається асинхронна функція `start`:

```
document.addEventListener("DOMContentLoaded", () => {
  const start = async() => {
```

На початку функції `start` створюється об'єкт `mindarThree` – екземпляр AR-рушія, реалізованого у бібліотеці MindAR, який розташує шари полотна та відео у тілі документа:

```
const mindarThree = new window.MINDAR.FACE.MindARThree({
  container: document.body,
});
```

Конструктор `window.MINDAR.FACE.MindARThree` створює рендерер, сцену та камеру `Three.js` – доступ до них можна отримати з відповідних властивостей об'єкту `mindarThree`:

```
const {renderer, scene, camera} = mindarThree;
```

Через те, що віртуальні аксесуари розташовані на обличчі, крім фонового освітлення `light` встановимо світильник (напрямлене світло) `light2` перед обличчям. Щоб світло від нього не било в очі, піднімемо його та змістимо вліво:

```
const light = new THREE.HemisphereLight(0xffffff, 0xbbbbff, 1);
const light2 = new THREE.DirectionalLight(0xffffff, 0.6);
light2.position.set(-0.5, 1, 1);
scene.add(light);
scene.add(light2);
```

Для того, щоб віртуальні аксесуари виглядали природно (зокрема, не були видимі тоді, коли реальна голова мала б їх закривати), створимо об'єкт-оклюдер – голову:

```
const occluder = await loadGLTF(
  "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/"+
  "examples/face-tracking/assets/sparkar/headOccluder.glb");
occluder.scene.scale.set(0.065, 0.065, 0.065);
occluder.scene.position.set(0, -0.3, 0.15);
```

Для того, щоб його не було видно, будемо його будувати у пам'яті без відображення на сцені:

```
occluder.scene.traverse((o) => {
  if (o.isMesh) {
    const occluderMaterial =
      new THREE.MeshPhongMaterial({colorWrite: false});
    o.material = occluderMaterial;
  }
});
```

Для правильної роботи алгоритмів кидання та трасування променів оклюдер має завжди будуватись першим:

```
occluder.scene.renderOrder = 0;
```

Створимо якір для оклюдера та прив'яжемо його до точки між очима (168):

```
const occluderAnchor = mindarThree.addAnchor(168);
occluderAnchor.group.add(occluder.scene);
```

До тієї ж точки прив'яжемо перші окуляри, проте відображати їх будемо завжди після оклюдера:

```
const glasses = await loadGLTF(
  "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/" +
  "examples/face-tracking/assets/glasses/scene.gltf");
glasses.scene.scale.set(0.01, 0.01, 0.01);
glasses.scene.renderOrder = 1;
const glassesAnchor = mindarThree.addAnchor(168);
glassesAnchor.group.add(glasses.scene);
```

Аналогічно завантажується та налаштовується друга модель окулярів:

```
const glasses2 = await loadGLTF(
  "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/" +
```

```

    "examples/face-tracking/assets/glasses2/scene.gltf");
glasses2.scene.rotation.set(0, -Math.PI/2, 0);
glasses2.scene.position.set(0, -0.3, 0);
glasses2.scene.scale.set(0.6, 0.6, 0.6);
glasses2.scene.renderOrder = 1;
const glasses2Anchor = mindarThree.addAnchor(168);
glasses2Anchor.group.add(glasses2.scene);

```

Обидва капелюхи прив'яжемо до найвищої точки маски обличчя (10):

```

const hat1 = await loadGLTF(
    "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/"+
    "examples/face-tracking/assets/hat/scene.gltf");
hat1.scene.position.set(0, 1, -0.5);
hat1.scene.scale.set(0.35, 0.35, 0.35);
hat1.scene.renderOrder = 1;
const hat1Anchor = mindarThree.addAnchor(10);
hat1Anchor.group.add(hat1.scene);

const hat2 = await loadGLTF(
    "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/"+
    "examples/face-tracking/assets/hat2/scene.gltf");
hat2.scene.position.set(0, -0.2, -0.5);
hat2.scene.scale.set(0.008, 0.008, 0.008);
hat2.scene.renderOrder = 1;
const hat2Anchor = mindarThree.addAnchor(10);
hat2Anchor.group.add(hat2.scene);

```

Хоча моделі сережок на ліве та праве вуха є ідентичними, прив'язуються вони до різних точок – 127 та 356:

```

const earringLeft = await loadGLTF(
    "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/"+
    "examples/face-tracking/assets/earring/scene.gltf");
earringLeft.scene.position.set(0, -0.3, -0.3);
earringLeft.scene.scale.set(0.05, 0.05, 0.05);

```

```

earringLeft.scene.renderOrder = 1;
const earringLeftAnchor = mindarThree.addAnchor(127);
earringLeftAnchor.group.add(earringLeft.scene);

const earringRight = await loadGLTF(
  "https://raw.githubusercontent.com/hiukim/mind-ar-js/master/" +
  "examples/face-tracking/assets/earring/scene.gltf");
earringRight.scene.position.set(0, -0.3, -0.3);
earringRight.scene.scale.set(0.05, 0.05, 0.05);
earringRight.scene.renderOrder = 1;
const earringRightAnchor = mindarThree.addAnchor(356);
earringRightAnchor.group.add(earringRight.scene);

```

Кожне із зображень у контейнері `selections` виступатиме у ролі кнопки, ідентифікатори яких з документу HTML заносяться до масиву `buttons`:

```

const buttons =
  ["#glasses1", "#glasses2", "#hat1", "#hat2", "#earring"];

```

Масив `models` містить завантажені моделі аксесуарів, а масив `visibles` визначає їх видимість:

```

const models = [
  [glasses.scene], [glasses2.scene], [hat1.scene], [hat2.scene],
  [earringLeft.scene, earringRight.scene]];
const visibles = [true, false, false, true, true];

```

Функція `setVisible` керуватиме видимістю кнопки вибору віртуального аксесуару та моделей. Для моделей достатньо встановити їх властивість `visible`, а для кнопок (що є зображеннями) видимість керується стилювим описом – наявністю чи відсутністю атрибуту `selected`:

```

const setVisible = (button, models, visible) => {
  if (visible) {

```

```

        button.classList.add("selected");
    } else {
        button.classList.remove("selected");
    }
models.forEach((model) => {
    model.visible = visible;
});
}

```

Приклад наявності декількох моделей, пов'язаних з одним аксесуаром – сережки для вух.

Дляожної кнопки з масиву `buttons` встановимо її початкову видимість та пов'язаних з нею моделей викликом `setVisible`:

```

buttons.forEach((buttonId, index) => {
    const button = document.querySelector(buttonId);
    setVisible(button, models[index], visibles[index]);
    button.addEventListener("click", () => {
        visibles[index] = !visibles[index];
        setVisible(button, models[index], visibles[index]);
    });
});

```

Отримаємо з документу HTML посилання на елементи попереднього перегляду захопленого зображення та його поширення:

```

const previewImage = document.querySelector("#preview-image");
const previewClose = document.querySelector("#preview-close");
const preview = document.querySelector("#preview");
const previewShare = document.querySelector("#preview-share");

```

При натисканні на кнопку захоплення фото виконуємо його захоплення викликом `capture`, робимо видимим зображення для попереднього перегляду встановленням відповідного стилювого елементу та пов'язуємо його із захопленим фото:

```
document.querySelector("#capture").addEventListener("click", ()=>{
    const data = capture(mindarThree);
    preview.style.visibility = "visible";
    previewImage.src = data;
});
```

При натисканні на кнопку закриття попереднього перегляду (X) приховуємо весь контейнер `preview`, тобто попереднє зображення, кнопки закриття та поширення:

```
previewClose.addEventListener("click", () => {
    preview.style.visibility = "hidden";
});
```

При натисканні на кнопку поширення (`Share`) копіюємо захоплене зображення до буферного полотна `canvas` та, за доступності інтерфейсу Web Share API, намагаємось поширити файл `photo.png` або, за недоступності, завантажити:

```
previewShare.addEventListener("click", () => {
    const canvas = document.createElement("canvas");
    canvas.width = previewImage.width;
    canvas.height = previewImage.height;
    const context = canvas.getContext("2d");
    context.drawImage(previewImage,
        0, 0, canvas.width, canvas.height);

    canvas.toBlob((blob) => {
        const file = new File([blob], "photo.png", {type: "image/png"});
        const files = [file];
        if (navigator.canShare && navigator.canShare({files})) {
            navigator.share({ files: files, title: "AR Photo", })
        } else {
            const link = document.createElement("a");
            link.download = "photo.png";
            link.href = previewImage.src;
```

```
    link.click();
}
});
});
});
```

Намагаємось ініціювати AR-рушій:

```
await mindarThree.start();
```

Налаштовуємо рендеринг сцени з точки зору камери для кожного ка-  
дру:

```
renderer.setAnimationLoop(() => {
  renderer.render(scene, camera);
});
}
```

Нарешті, викликаємо функцію **start**, що визначалась останні сторінки:

```
start();
});
```

## 5 Відстеження довкілля

### 5.1 Початок роботи із WebXR

WebXR Device API (WebXR) – це інтерфейс програмування веб-додатків, який описує підтримку доступу до пристройів доповненої реальності та віртуальної реальності, таких як HTC Vive, Oculus Rift, Oculus Quest, Google Cardboard, HoloLens, Magic Leap або Open Source Virtual Reality (OSVR), у веб-браузері (<https://immersiveweb.dev/>). WebXR та пов'язані з ним API є стандартами, тому їх застосування можливе без використання будь-яких додаткових бібліотек у будь-якому браузері.

Наразі достатньо повну підтримку WebXR забезпечує Chrome на пристроях під управлінням Android, а з червня 2022 року експериментальна підтримка доступна у Safari на пристроях під управлінням iOS – на старих пристроях у нагоді стане WebXR Viewer (<https://apps.apple.com/us/app/webxr-viewer/id1295998056>).

Для розробки WebXR додатків доцільним є встановлення розширення Chrome WebXR API Emulator (<https://chrome.google.com/webstore/detail/webxr-api-emulator/mjddjgeghkdijejnaciaefnkjmkafnnej/related?hl=en>). На рис. 5.1 показано вкладку WebXR, що з'являється після встановлення розширення, з можливістю вибору емульованого пристрою та його параметрів (<https://blog.mozvr.com/webxr-emulator-extension/>).

Для тестування WebXR API створимо документ HTML, у якому відсутнє підключення бібліотеки MindAR – потреби у ній немає, тому що, по-перше, вона не підтримує відстеження довкілля (на відміну, наприклад, від 8th Walls), а, по-друге, усі дії у доповненій реальності будемо виконувати за допомогою лише WebXR. Через те, що наразі не всі пристройі підтримують цей стандарт, для початку роботи створимо кнопку із надписом “Увійти до WebXR”, яку розташуємо у центрі нижньої частини документу:

```
<html>
  <head>
    <meta name="viewport"
```

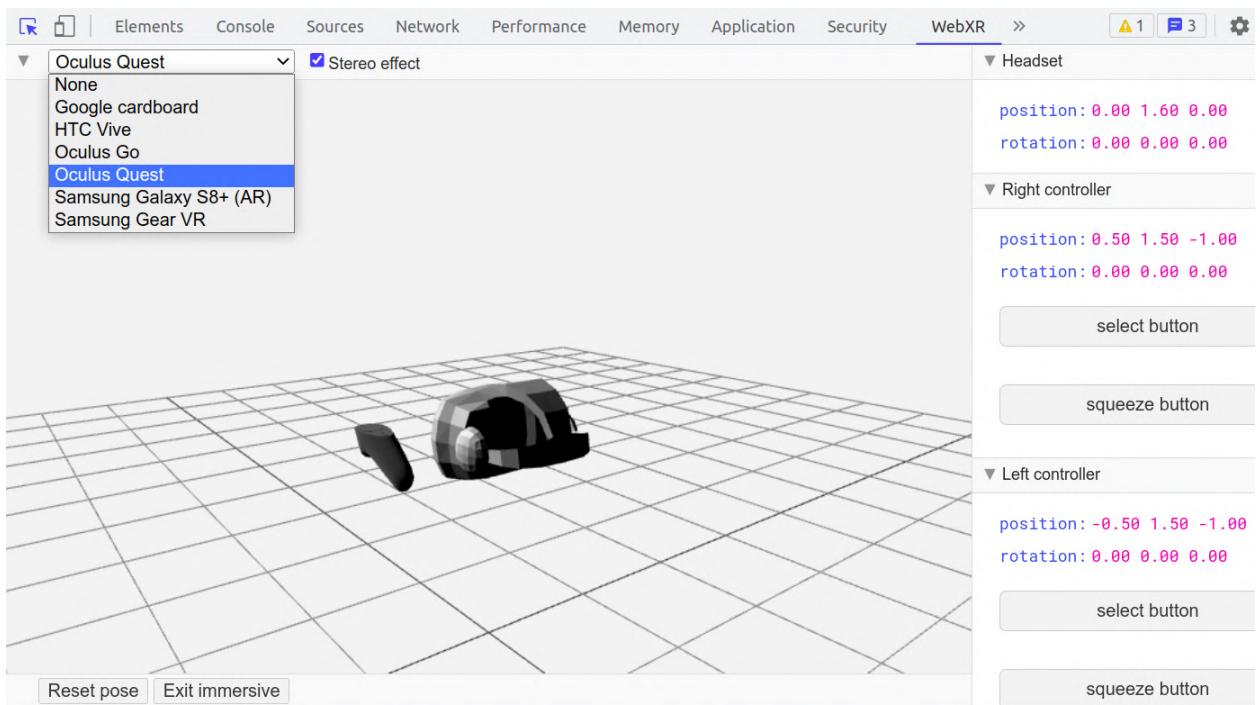


Рис. 5.1. Налаштування WebXR API Emulator.

```

        content="width=device-width, initial-scale=1.0">
<script type="module" src="./main.js"></script>
<style>
    body {margin: 0}
    #ar-button {position: fixed; z-index: 1001; padding: 1vh;
    bottom: 1vh; left: 50%; transform: translateX(-50%)}
</style>
</head>
<body>
    <button id="ar-button">Увійти до WebXR</button>
</body>
</html>

```

Структура програмного коду досить традиційна – усі дії виконуватимуться після того, як документ завантажений. Підключення бібліотеки Three.js виконується для того, щоб у асинхронній функції `initialize` (раніше використовували для неї ім’я `start`) створити об’єкт доповненої реальності:

```
import * as THREE from
```

```

"https://unpkg.com/three/build/three.module.js";

document.addEventListener("DOMContentLoaded", () => {
    const initialize = async() => {
        // код функції initialize
    }

    initialize();
}) ;

```

`arButton` є об'єктом DOM класу `button` з ідентифікатором `ar-button` – зробимо цю кнопку квазібагатофункціональною шляхом зміни надпису на ній:

```
const arButton = document.querySelector("#ar-button");
```

Доступ до WebXR надається тільки в безпечному контексті (HTTPS). Властивість `xr` інтерфейсу `Navigator`, доступна тільки для читання, повертає об'єкт `XRSystem`, який може бути використаний для доступу до WebXR Device API. Кожне вікно (`Window`) має власний екземпляр інтерфейсу `Navigator`, до якого можна отримати доступ як `window.navigator` або як `navigator`. При цьому також створюється новий екземпляр `XRSystem`, який приєднується до екземпляру навігатора як `navigator.xr`. Якщо властивість `xr` існує, то її можна використовувати для доступу до WebXR Device API.

Метод `XRSystem isSessionSupported` повертає проміс, тому й викликається як асинхронний. Значення промісу перетворюється на `true`, якщо вказаний режим сеансу WebXR підтримується пристроєм користувача. Якщо пристрій відсутній або браузер не має дозволу на використання XR-пристрою, повертається значення `false`.

Параметром `isSessionSupported` є рядок, що визначає режим сеансу WebXR, для якого повинна бути перевірена підтримка: `immersive-ar`, `immersive-vr` або `inline`.

Згідно документації (<https://developer.mozilla.org/en-US/docs/Web/API/Navigator/xr>), на 9 вересня 2022 року повна підтримка WebXR

реалізована у веб-браузерах Chrome, Edge, Opera, Samsung Internet – на них буде доступна властивість `xr` інтерфейсу `Navigator`. Проте, чи буде обраний режим WebXR підтримуватись, залежить також від апаратного забезпечення та дозволів, що їх надає користувач. Таким чином, змінна `supported` приймає істинне значення тоді, коли браузер підтримує WebXR та є можливість розпочати сеанс WebXR у доповнений реальності:

```
const supported = navigator.xr &&
    await navigator.xr.isSessionSupported("immersive-ar");
```

Якщо режим WebXR не підтримується, кнопка відключається, а надпис на ній змінюється на відповідний:

```
if (!supported) {
    arButton.textContent = "WebXR не підтримується";
    arButton.disabled = true;
    return;
}
```

Наступним кроком є створення кубу зеленого кольору `mesh` розміром  $6 \times 6 \times 6$  см (0.06 м – саме метр у фізичному світі є одиницею вимірювання у WebXR). Властивість `position` кубу вказує його положення у 30 см перед стартовим положенням, у якому знаходиться камера `camera`. Попри те, що матеріал кубу `MeshBasicMaterial` завжди має власний колір і не взаємодіє із напівсферичним освітленням `light`, для подальшої роботи (наприклад, завантаження моделей) освітлення є необхідним. Куб `mesh` і джерело світла `light` розташовуються на сцені `scene`. Побачити її з позиції камери `camera` (за замовчанням вона у початку координат) можна за допомогою рендерера `renderer`, при створенні якого включено згладжування (`antialias`) та прозорість (`alpha`). Виклик `appendChild` додає до тіла документа пов'язане із рендерером полотно, розмір якого викликом `setSize` встановлений у розмір вікна. Нарешті, для подолання ефекту замілювання полотна розмір пікселя полотна встановлений рівним розміру екранного пікселя:

```
const scene = new THREE.Scene();
```

```

const camera = new THREE.PerspectiveCamera();

const renderer = new THREE.WebGLRenderer({
    antialias: true, alpha: true
});
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.setPixelRatio(window.devicePixelRatio);
document.body.appendChild(renderer.domElement);

const geometry = new THREE.BoxGeometry(0.06, 0.06, 0.06);
const material = new THREE.MeshBasicMaterial({color: 0x00ff00});
const mesh = new THREE.Mesh(geometry, material);
mesh.position.set(0, 0, -0.3);
scene.add(mesh);

const light = new THREE.HemisphereLight(0xffffff, 0xbfff00, 1);
scene.add(light);

```

Відстежити початок та кінець сесії WebXR можна, опрацювавши події `sessionstart` та `sessionend` – нижче це демонструється:

```

renderer.xr.addEventListener("sessionstart", (e) => {
    console.log("Сесію WebXR розпочато");
});

renderer.xr.addEventListener("sessionend", () => {
    console.log("Сесію WebXR завершено");
});

```

Функція `start` ініціюватиме сеанс (сесію) WebXR:

```

let currentSession = null;
const start = async() => {

```

Інтерфейс `XRSession` WebXR Device API представляє поточний сеанс XR `currentSession`, надаючи методи та властивості, що використовуються для взаємодії з сеансом та керування ним. За допомогою методів `XRSession`

можна опитувати положення та орієнтацію глядача (`XRViewerPose`), збирати інформацію про оточення користувача та представляти зображення користувачеві.

Щоб відкрити сеанс WebXR, використовується асинхронний метод інтерфейсу `XRSysTem` (повертає проміс) `requestSession`, першим параметром якого є режим WebXR:

- `immersive-ar` – сеансу буде наданий повний доступ до імерсивного пристрою, але відрендерений вміст буде скомбінований із реальним оточенням;
- `immersive-vr` – результат рендерингу буде відображатися за допомогою імерсивного XR-пристрою в режимі віртуальної реальності;
- `inline` – результат рендерингу буде вбудовано у контекст елементу стандартного документу HTML. Убудовані сеанси можуть бути представлені в моно або стерео режимі, можуть мати або не мати відстеження глядача та не вимагають спеціального апаратного забезпечення.

Другий параметр `requestSession` є об'єктом для налаштування `XRSession`, що може містити поля `requiredFeatures` (масив значень, які сесія має підтримувати), `optionalFeatures` (масив значень, що ідентифікують функції, які сесія може підтримувати), `domOverlay` (об'єкт з обов'язковою кореневою властивістю, яка визначає елемент накладання, що буде відображатися користувачеві як вміст DOM-оверлею), `depthSensing` (об'єкт із двома обов'язковими властивостями `usagePreference` та `dataFormatPreference` для налаштування способу вимірювання глибини зображення).

Повний перелік властивостей сесії WebXR доступний у документації ([https://developer.mozilla.org/en-US/docs/Web/API/XRSysTem/requestSession#session\\_features](https://developer.mozilla.org/en-US/docs/Web/API/XRSysTem/requestSession#session_features)). У прикладі встановлено, що тіло документа HTML є DOM-оверлеєм – це необхідно для того, щоб розмістити кнопку `arButton` поверх вікна:

```
currentSession = await navigator.xr.requestSession(
```

```

    "immersive-ar", {
      optionalFeatures: ["dom-overlay"] ,
      domOverlay: {root: document.body}
    }
);

```

Three.js підтримує WebXR Device API через **WebXRManager** – внутрішній інтерфейс класу **WebGLRenderer**, доступ до якого дається через властивість **xr** об'єкту **renderer**. Прапорець **enabled** повідомляє рендереру про готовність до рендерингу XR – це дуже схоже на те, як працює MindAR або інші AR рушії: **WebXRManager** допомагає постійно оновлювати позицію сценів і камери так, щоб об'єкти виглядали прив'язаними до фізичного місця:

```
renderer.xr.enabled = true;
```

Метод **setReferenceSpaceType** визначає тип опорного простору, який потрібно встановити. Це метод може бути використаний для налаштування просторового зв'язку з фізичним оточенням користувача. Залежно від того, як користувач рухається в 3D просторі, встановлення відповідного опорного простору може покращити відстеження (за замовчуванням – **local-floor**):

- **local** – простір відстеження, розташований поблизу позиції глядача під час створення сеансу: очікується, що користувач не буде сильно переміщатися, якщо взагалі буде, за межі своєї початкової позиції, і відстеження оптимізовано для цього випадку використання;
- **bounded-floor** – простір відстеження, подібний до типу **local**, за винятком того, що користувач не повинен виходити за межі заздалегідь визначеної межі, заданої властивістю **boundsGeometry**;
- **local-floor** – простір відстеження подібний до **local**, за винятком того, що початкова позиція розміщується в безпечному для глядача місці, де значення осі у дорівнює 0 на рівні підлоги;
- **unbounded** – простір відстеження, який дозволяє користувачеві повну свободу пересування, можливо, на надзвичайно великих відстанях від

початкової точки: відстеження оптимізовано для стабільності навколо поточного положення користувача, тому початкова точка може зміщуватись в міру необхідності, щоб задоволити цю потребу;

- `viewer` – простір відстеження положення та орієнтації глядача: це особливо корисно при визначенні відстані між глядачем та об'єктом.

Для доповненої реальності вибір простору не дуже суттєвий, тому вибір `local` є чи не єдиним, що має сенс:

```
renderer.xr.setReferenceSpaceType("local");
```

Three.js не має можливості відстеження об'єктів, тому застосовується метод `setSession`, який передає Three.js інформацію про сесію WebXR:

```
await renderer.xr.setSession(currentSession);
```

Останні дії, що виконуються у функції `start` – зміна надпису на кнопці та запуск циклу анімації:

```
arButton.textContent = "Завершити сесію WebXR";
```

```
renderer.setAnimationLoop(() => {
  renderer.render(scene, camera);
});
```

Функція `end` завершує сесію WebXR викликом методу `end` інтерфейсу `XRSession`. Цикл анімації, в якому викликається метод `render`, зупиняється його заміною на порожній (`null`), полотно очищується викликом `clear`, а кнопка приховується:

```
const end = async() => {
  currentSession.end();
  renderer.setAnimationLoop(null);
  renderer.clear();
  arButton.style.display = "none";
}
```

Обробник події `click` натискання на кнопку запускає сесію WebXR, якщо вона не була запущена раніше, та припиняє у протилежному випадку – це надає можливість входити до режиму WebXR не автоматично, а за запитом користувача:

```
arButton.addEventListener("click", () => {
  if (currentSession) {
    end();
  } else {
    start();
  }
});
```

На рис. 5.2 та 5.3 показано роботу програмного забезпечення у режимі емулятора WebXR та на пристрой з підтримкою WebXR.

## 5.2 Застосування компоненту ARButton

Попередній приклад був досить довгим, і найбільша його частина була присвячена управлінню життєвим циклом імерсивної AR сесії. Зокрема, була створена кнопка для запуску та завершення сесії, перевірено, чи підтримує браузера WebXR тощо. Це достатньо стандартні речі, які реалізовані у додатковому компоненті Three.js – модулі `ARButton`. Якщо зміритись із тим, що надписи на кнопках є фіксованими, а зображення куба на полотні з'являється до початку сесії WebXR (рис. 5.4), можна суттєво спростити код як документу HTML (в якому вже немає потреби створювати кнопку та визначати її стиль), так її основної програми:

```
import * as THREE from
  "https://unpkg.com/three/build/three.module.js";
import {ARButton} from
  "https://unpkg.com/three/examples/jsm/webxr/ARButton.js";

document.addEventListener("DOMContentLoaded", () => {
  const initialize = async() => {
    const scene = new THREE.Scene();
```

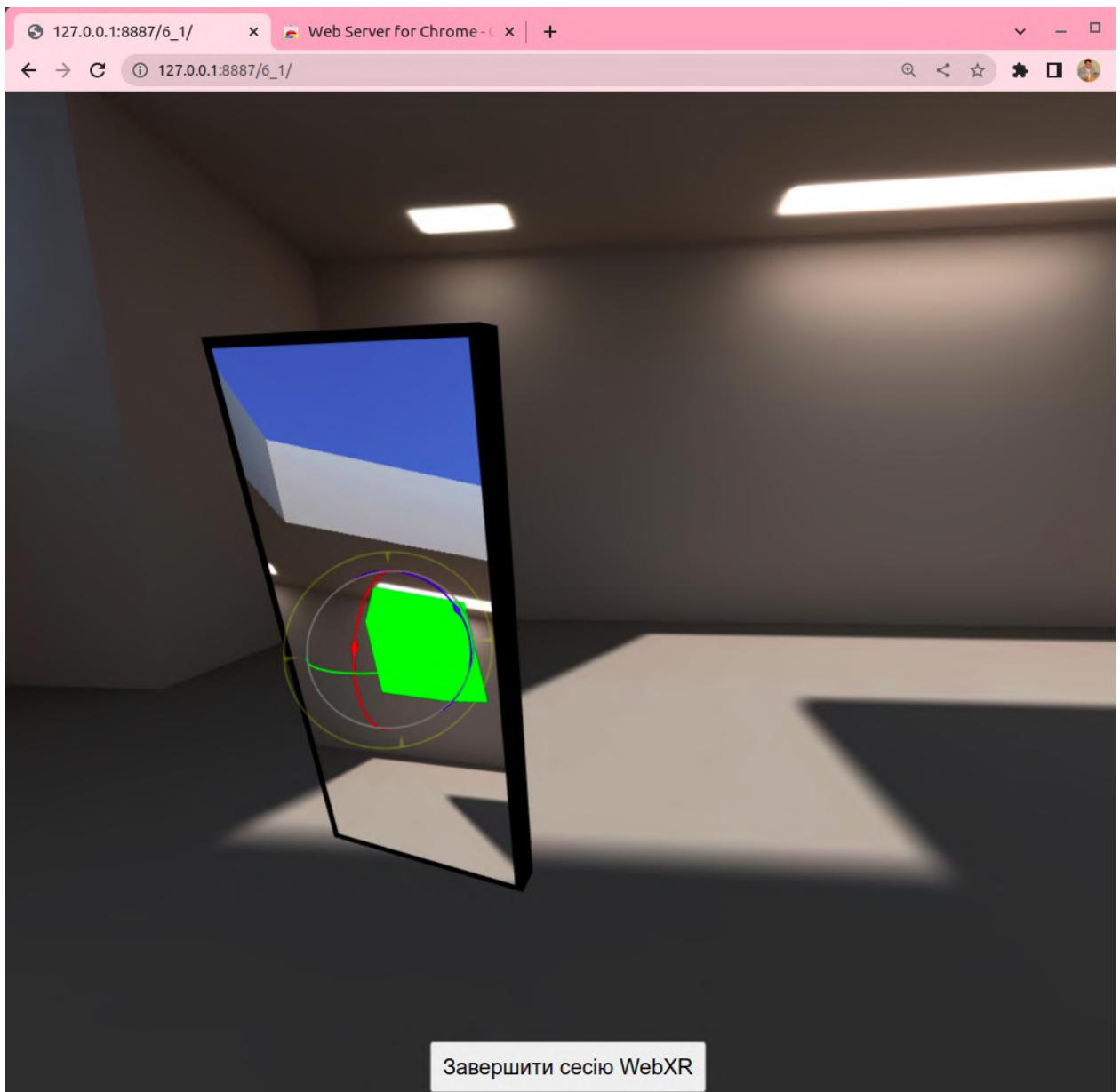


Рис. 5.2. Тестування програмного забезпечення у режимі емулятора WebXR.

```
const camera = new THREE.PerspectiveCamera();  
  
const renderer = new  
    THREE.WebGLRenderer({antialias: true, alpha: true});  
renderer.setSize(window.innerWidth, window.innerHeight);  
renderer.setPixelRatio(window.devicePixelRatio);  
document.body.appendChild(renderer.domElement);  
  
const geometry = new THREE.BoxGeometry(0.06, 0.06, 0.06);
```

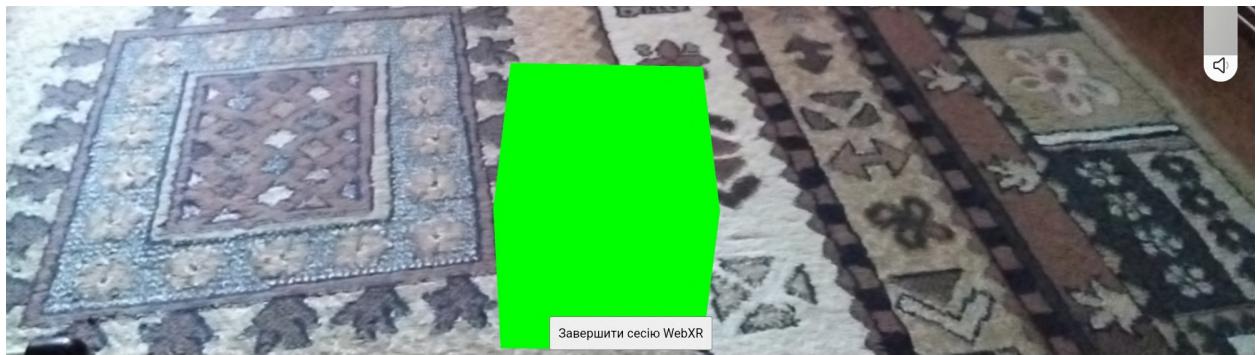


Рис. 5.3. Тестування програмного забезпечення на пристрой з підтримкою WebXR.

```
const material = new THREE.MeshBasicMaterial({color: 0x00ff00});
const mesh = new THREE.Mesh(geometry, material);
mesh.position.set(0, 0, -0.3);
scene.add(mesh);

const light = new THREE.HemisphereLight(0xffffff, 0xbbbbff, 1);
scene.add(light);

renderer.xr.enabled = true;
renderer.setAnimationLoop(() => {
    renderer.render(scene, camera);
});

const arButton = ARButton.createButton(renderer,
{
    optionalFeatures: ["dom-overlay"],
    domOverlay: {root: document.body}
});
document.body.appendChild(arButton);

}

initialize();
});
```

Підключення модуля **ARButton** виконується окремою командою **import**,



Рис. 5.4. Тестування кнопки з компоненту ARButton.

адже він не є стандартною складовою бібліотеки Three.js. Єдині подальші нагадування про те, що це не лише код для створення куба – рядки із вказівкою рендереру застосовувати WebXR (встановленням `renderer.xr.enabled`) та виклик методу `createButton` з підключеним модулем. Параметрами `createButton` є рендерер та додаткові параметри, що передавались у попередньому прикладі до `navigator.xr.requestSession`.

Створена кнопка додається до тіла документу традиційним викликом `appendChild`.

## 5.3 Управління контроллерами

Повноцінний досвід роботи з WebXR – це не просто показ користувачеві віртуальної сцени або доповнення реальності: користувач повинен мати можливість взаємодіяти з ним. З цією метою WebXR забезпечує підтримку різних типів пристройів введення.

Вхідні дані у WebXR діляться на дві основні категорії: наведення та дії. Наведення (націлювання) – це визначення точки в просторі: натискання користувачем на точку на екрані, відстеження його очей або використання джойстика чи контроллера, що реагує на рух, для переміщення курсору. Дії включають як дії вибору, такі як натискання на кнопку, так і дії натискання, такі як натискання на спусковий гачок або захоплення об'єкту при використанні тактильних рукавичок.

WebXR підтримує такі основні типи пристройів для опрацювання подій наведення та дій:

- *дотики до екрану* (зокрема, на телефонах або планшетах) можуть використовуватися для одночасного наведення на ціль і вибору;
- *контролери із датчиками руху*, які використовують акселерометри, магнітотетри та інші датчики для відстеження руху та наведення на ціль та можуть додатково включати будь-яку кількість кнопок, джойстиків, накладок для великих пальців, сенсорних панелей, датчиків сили тощо для забезпечення додаткових джерел введення як для наведення на ціль, так і для вибору;
- *тригери, що стискаються* (зокрема, рукавички), для забезпечення дії стискання;
- *голосові команди* з використанням розпізнавання мови;
- *відстеження погляду* (слідування за рухами очей для вибору цілей) тощо.

Дії користувача у WebXR фіксуються та опрацьовуються за допомогою контроллерів. У Three.js метод `getController` класу `WebXRManager` повертає групу так званого цільового променевого простору контроллера XR (`XRSpace`), який можна використовувати для візуалізації 3D об'єктів, що

підтримують просторовий користувацький інтерфейс. Початок цього простору знаходиться в точці, з якої випромінюється промінь цілі (наприклад, передній кінець контролера або кінець ствола гармати, якщо контроллер візуалізується як гармата), а вектор орієнтації простору простягається назовні вздовж траєкторії променя цілі.

Контролери наразі більше застосовуються у пристроях та додатках віртуальної реальності – для доповненої реальності основним контроллером є сенсорний екран мобільного пристрою, доступ до якого можна отримати у такий спосіб:

```
const controller = renderer.xr.getController(0);
```

Контроллер з індексом 0 як раз й відповідає першому сенсорному екрану мобільного пристрою. Це пов'язано із тим, що в імерсивному досвіді навігація у віртуальному середовищі відбувається за допомогою мобільного пристрою, тому мобільний пристрій виконує роль контроллера.

Кожне джерело вхідних даних повинно визначати первинну дію. Первинна дія, або дія вибору – це специфічна для платформи дія, яка реагує на маніпуляції користувача, доставляючи, по порядку, такі події:

1. Подія `selectstart` вказує на те, що користувач виконав дію, яка запускає первинну дію. Це може бути жест, натискання кнопки тощо.
2. Якщо первинна дія завершується успішно (наприклад, через те, що користувач відпустив кнопку або тригер), а не через помилку, то відправляється подія `select`.
3. Після відправлення події `select` або якщо контролер, на якому виконується дія, відключається чи іншим чином стає недоступним, відправляється подія `selectend`.

Відповідні події стискання (`squeezestart`, `squeezeend` та `squeeze`) на такому контроллері, як сенсорний екран, реалізовувати недоцільно.

Для демонстрації опрацювання подій `selectstart`, `select` та `selectend` додамо до документу HTML поле багаторядкового текстового редактору, до якого будемо спрямовувати повідомлення про події (рис. 5.5):

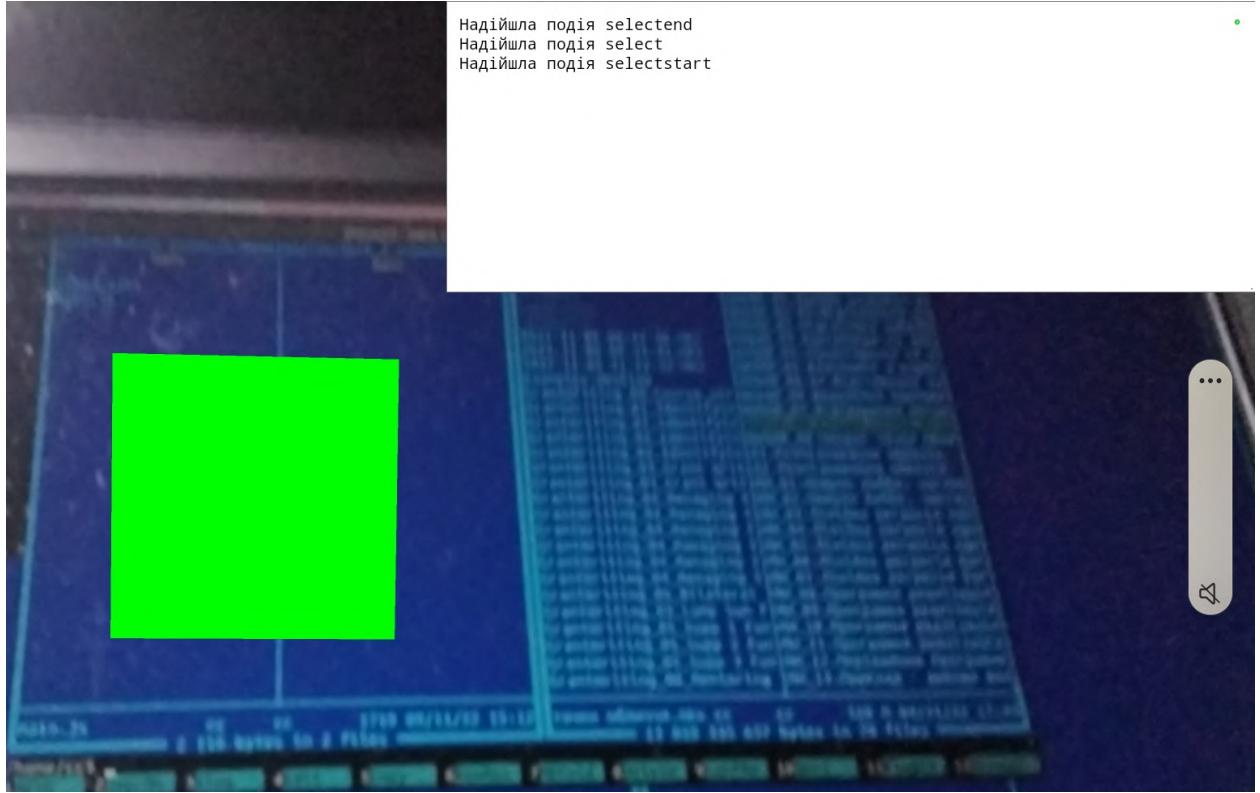


Рис. 5.5. Демонстрація роботи сенсорного екрану як контроллеру.

```
<textarea id="events"></textarea>
```

із такими стилювими налаштуваннями:

```
#events {  
    position: absolute; right: 0; top: 0; height: 30vh; padding: 10px;  
    width: 50vw; overflow: auto; background: white; z-index: 1001;  
}
```

У програмному коді після створення контроллеру додамо обробники подій, за якими до текстового редактору будуть додаватись рядки:

```
const eventsDiv = document.querySelector("#events");  
  
controller.addEventListener('selectstart', () => {  
    eventsDiv.prepend("Надійшла подія selectstart\n");  
});  
controller.addEventListener('selectend', () => {  
    eventsDiv.prepend("Надійшла подія selectend\n");
```

```

}) ;

controller.addEventListener('select', () => {
    eventsDiv.prepend("Надійшла подія select\n");
}) ;

```

Звичайно, можна було б вивести ці повідомлення до консолі браузера викликом `console.log`, проте її перегляд на мобільному пристрої часто є утрудненим (при роботі в емуляторі ці події можна ініціювати правою кнопкою миші на зображені мобільного пристроя).

Опрацювання подій `selectstart`, `select` та `selectend` є подібним до класичних сенсорних подій `touchstart`, `touchmove`, `touchcancel` та `touchend`, проте події контроллера та сенсорні події не є ідентичними: події контроллера оперують 3D координатами імерсивного простору, а сенсорні події – 2D координатами на сенсорному екрані.

## 5.4 Розміщення об'єктів

Застосуємо подію від контроллера `select` для розміщення об'єктів у довкіллі (рис. 5.6). Програмний код розпочинається із підключення бібліотеки `Three.js` та модуля `ARButton`:

```

import * as THREE
from "https://unpkg.com/three/build/three.module.js";
import {ARButton}
from "https://unpkg.com/three/examples/jsm/webxr/ARButton.js";

```

Після завантаження документа створюємо та викликаємо функцію `initialize`:

```

document.addEventListener("DOMContentLoaded", () => {
    const initialize = async() => {
        //вміст функції
    }
    initialize();
}) ;

```

Усі об'єкти розміщуватимуться на сцені:

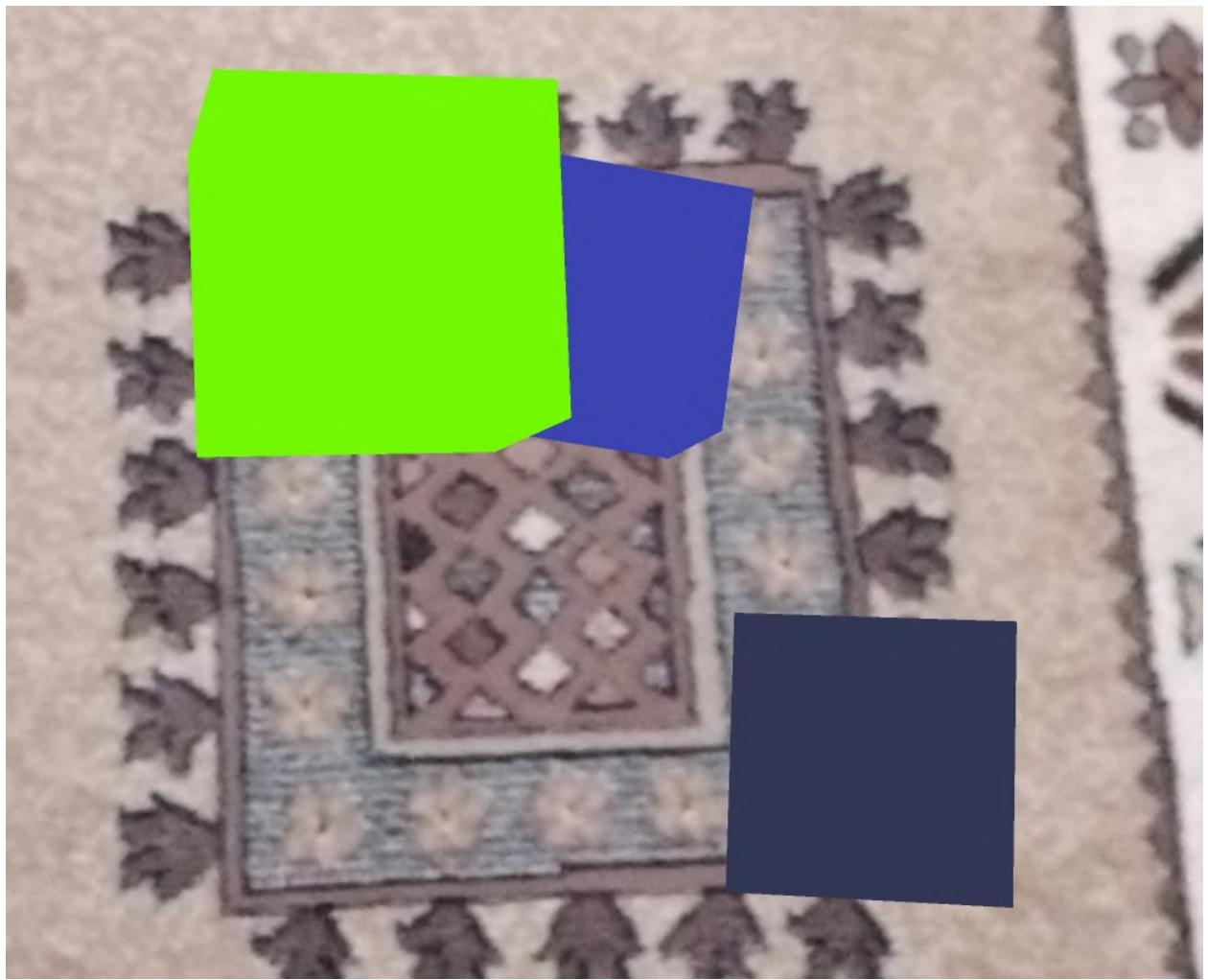


Рис. 5.6. Демонстрація динамічного розміщення об'єктів у довкіллі.

```
const scene = new THREE.Scene();
```

Видимий розмір об'єктів визначається камерою – вона налаштована так, щоб співвідношення сторін її об'ективу було таким самим, як у вікна браузера, розтруб камери встановлений у  $70^\circ$ , а видимими будуть лише об'єкти, що знаходяться на відстані від нього не ближче 1 см та не далі 20 м:

```
const camera = new THREE.PerspectiveCamera(70,
    window.innerWidth / window.innerHeight, 0.01, 20);
```

Об'єкти на сцені будуть освітлюватись напівсферичним освітленням:

```
const light = new THREE.HemisphereLight(0xfffffff, 0xbfffff, 1);
scene.add(light);
```

Для їх кращого відображення рендерером включено згладжування та прозорість, а форма пікселя полотна встановлена у відповідну форму пікселя пристрою:

```
const renderer = new
    THREE.WebGLRenderer({antialias: true, alpha: true});
renderer.setPixelRatio(window.devicePixelRatio);
```

Розмір полотна відповідає розміру вікна браузера:

```
renderer.setSize(window.innerWidth, window.innerHeight);
```

Повідомляємо рендереру про готовність до рендерингу у WebXR:

```
renderer.xr.enabled = true;
```

Налаштовуємо цикл анімації:

```
renderer.setAnimationLoop(() => {
    renderer.render(scene, camera);
});
```

Застосовуємо модуль ARButton для ініціалізації WebXR за натисканням на кнопку:

```
const arButton = ARButton.createButton(renderer, {
    optionalFeatures: ["dom-overlay"],
    domOverlay: {root: document.body}
});
document.body.appendChild(renderer.domElement);
document.body.appendChild(arButton);
```

Створюємо об'єкт для управління контроллером (у даному випадку – сенсорним екраном) та додаємо його до сцени – це надасть можливість визначати його координати на сцені:

```
const controller = renderer.xr.getController(0);
scene.add(controller);
```

Найважливішою частиною є обробник події `select` – натискання на сенсорний екран:

```
controller.addEventListener("select", () => {
    const geometry = new THREE.BoxGeometry(0.06, 0.06, 0.06);
    const material = new
        THREE.MeshBasicMaterial({color: 0xffffffff * Math.random()});
    const mesh = new THREE.Mesh(geometry, material);
    mesh.position.applyMatrix4(controller.matrixWorld);
    mesh.quaternion.setFromRotationMatrix(controller.matrixWorld);
    scene.add(mesh);
});
```

Щойно від контроллера надходить ця подія, на сцену додається новий об'єкт – куб довільного кольору. На жаль, просто скопіювати вектори координат та кутів нахилу контроллеру до відповідних складових кубічного об'єкту недостатньо.

Найпоширенішим використання кватерніонів та матриць у тривимірній комп'ютерній графіці є *кватерніони обертання* та *матриця перетворення*  $4 \times 4$ . За допомогою перемноження з матрицею переходу стають можливими перетворення тривимірного вектора, що являє точку в тривимірному просторі, як-от перенесення, обертання, зсув, масштабування, відображення, ортогональне або перспективне проектування тощо. Це називається застосуванням матриці до вектора.

Кожен тривимірний об'єкт Three.js (`Object3D` та його нащадки) містить три взаємопов'язані матриці:

- `matrix` – матриця локального перетворення об'єкта;
- `matrixWorld` – матриця глобального (“світового”) перетворення об'єкта щодо його батька (якщо в об'єкта немає батьківського, воно ідентична матриці локального перетворення);
- `modelViewMatrix` – матриця перетворення об'єкта щодо системи координат камери (матриця `matrixWorld`, помножена на матрицю `matrixWorldInverse` камери).

Метод `applyMatrix4` застосовує до вектора `position` кубу `mesh` матрицю глобального перетворення `matrixWorld` контроллера `controller`, у результаті чого оновлюється положення, кут повороту та масштаб кубу.

Метод `setFromRotationMatrix` виконує поворот кватерніону `quaternion` кубу `mesh` (локальний кут повороту об'єкта) за компонентом матриці обертання `matrixWorld` контроллера `controller`.

## 5.5 Перевірка дотику

Перевірка дотику (*hit testing*, жарг. *реальнний хіт-тестинг*), як це розуміється у <https://www.w3.org/TR/webxr-hit-test-1/>, є актом перевірки того, чи перетинається ідеалізований математичний промінь (напівління) з *реальним світом*, як це розуміється базовим апаратним та програмним забезпеченням доповненої реальності. Перетин променів з *віртуальними об'єктами* не входить в сферу застосування WebXR Hit Test API та відноситься до рейткастингу (віртуального хіт-тестингу).

Найчастіше у WebXR розробники проводять перевірку дотику, використовуючи `XRIInputSources` або `XRReferenceSpace` типу `viewer`, щоб відстежити, куди слід виконати наведення на мобільних пристроях, або для того, щоб розташувати віртуальний об'єкт у геометрії реального світу. У сесіях WebXR типу `inline` та `immersive-vr` обмежуються виконанням віртуальних хіт-тестів, тоді як у сесіях `immersive-ar` можуть виконувати як віртуальні, так і реальні хіт-тести.

WebXR Hit Test API використовується насамперед для:

- відображення об'єкта, що відстежує реальні поверхні, на які спрямований пристрій або контроллер (наприклад, його “прилипання” до реальних поверхонь, коли користувач переміщує контроллер);
- розміщення віртуальних об'єктів у реальному світі так, щоб вони виглядали закріпленими у реальному світі – для цього вони повинні бути розміщені на тій самій висоті, що й об'єкти реального світу (підлога, стіл, стіна тощо).

Хіт-тестинг є дуже потужною концепцією, але його реалізація на різних пристроях поки що є повною, тому наразі він переважно використо-

вується для виявлення поверхонь – поверхні столу, поверхні підлоги, поверхні стіни тощо. Та навіть це надає можливість суттєво підвищити реалістичність розміщення віртуальних об'єктів у довкіллі.

Через значні обчислювальні витрати перевірка дотику за замовчанням не виконується – її потрібно запросити явно у `navigator.xr.requestSession`, вказавши `hit-test` у параметрі `requiredFeatures`:

```
currentSession = await navigator.xr.requestSession("immersive-ar", {  
    requiredFeatures: ["hit-test"],  
    optionalFeatures: ["dom-overlay"],  
    domOverlay: {root: document.body}  
});
```

При використанні `ARButton` передавання параметрів виконується аналогічно:

```
import * as THREE from  
  "https://unpkg.com/three/build/three.module.js";  
import {ARButton} from  
  "https://unpkg.com/three/examples/jsm/webxr/ARButton.js";  
  
document.addEventListener('DOMContentLoaded', () => {  
  const initialize = async() => {  
    const scene = new THREE.Scene();  
    const camera = new THREE.PerspectiveCamera();  
  
    const light = new THREE.HemisphereLight(0xffffff, 0xbbbbbff, 1);  
    scene.add(light);  
  
    const reticleGeometry = new THREE.RingGeometry(0.15, 0.2, 32);  
    reticleGeometry.rotateX(- Math.PI / 2);  
    const reticleMaterial = new THREE.MeshBasicMaterial();  
    const reticle = new THREE.Mesh(reticleGeometry, reticleMaterial);  
    reticle.matrixAutoUpdate = false;  
    reticle.visible = false;
```

```

scene.add(reticle);

const renderer = new THREE.WebGLRenderer({
    antialias: true, alpha: true
});
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.xr.enabled = true;

const arButton = ARButton.createButton(renderer, {
    requiredFeatures: ['hit-test'],
    optionalFeatures: ['dom-overlay'],
    domOverlay: {root: document.body}
});
document.body.appendChild(renderer.domElement);
document.body.appendChild(arButton);

const controller = renderer.xr.getController(0);
scene.add(controller);

```

До попереднього прикладу внесені також такі зміни:

- створення камери виконано викликом `PerspectiveCamera` із параметрами за замовчаннями: вона налаштована так, щоб співвідношення сторін її об'єктиву було 1:1 (квадратне полотно), розтруб камери встановлений у  $50^\circ$ , а видимими будуть лише об'єкти, що знаходяться на відстані від нього не біжче 10 см та не далі 2 км;
- до сцени додано правильний тридцятидводрігнник `reticle` білого кольору зовнішнім радіусом 20 см, а внутрішнім – 15 см, тобто кільце: через те, що цей об'єкт спочатку приховується, для того, щоб кожного кадру не виконувати обчислення зміни його геометрії у залежності від зміни світових координат (`matrixWorld`), параметр `matrixAutoUpdate` вимкнено.

Кільце `reticle` буде індикаторним об'єктом – своєрідним прицілом,

який відображається та оновлює координати у місці дотику: для того, щоб він завжди лежав на поверхні дотику, й виконується поворот кільця на 90°.

У обробнику події `select` куб `mesh` розташовується тепер не у позиції контроллера (`controller.matrixWorld`), а у позиції кільця `reticle`. Прибирання виклику методу `setFromRotationMatrix` призводить до того, що куб `mesh` не повертається згідно орієнтації пристрою – тепер його орієнтація збігається з орієнтацією кільця. Певне розмаїття вносить випадкова зміна висоти кубу від 1 до 3 разів, тобто перетворення його на прямокутний паралелепіпед:

```
controller.addEventListener('select', () => {
    const geometry = new THREE.BoxGeometry(0.06, 0.06, 0.06);
    const material = new THREE.MeshBasicMaterial({
        color: 0xffffffff * Math.random()
    });
    const mesh = new THREE.Mesh(geometry, material);
    mesh.position.setFromMatrixPosition(reticle.matrix);
    mesh.scale.y = Math.random() * 2 + 1;
    scene.add(mesh);
});
```

Кільце `reticle` виконуватиме роль мітки на поверхні (рис. 5.7), дотик до якої перевірятиметься за допомогою WebXR Hit Test API – це можна налаштувати після початку сесії (сеансу) WebXR у обробнику події `sessionstart`:

```
renderer.xr.addEventListener("sessionstart", async (e) => {
    // зміст обробника події початку сесії WebXR
});
```

Для того, щоб опрацювати подію `sessionstart`, спочатку необхідно отримати відомості про поточну сесію WebXR, приховані у модулі `ARButton`. У `Three.js` властивість `xr` об'єкту `renderer` є об'єктом класу `WebXRManager` – абстракції WebXR Device API. Метод `getSession` цього класу повертає об'єкт `session` класу `XRSession`, необхідний для управління активними сесіями WebXR:



Рис. 5.7. Тестування модуля WebXR Hit Test на різних типах поверхонь.

```
const session = renderer.xr.getSession();
```

Модуль `ARButton` встановлює тип опорного простору у `local`, що відповідає довкіллю користувача, який мало рухається – координатна система розташовується навколо нього та не зміщується. Для нашого випадку більш доцільним буде вибір опорного простору `viewer`, в якому відстежується положення та орієнтація глядача, що змінюються. Зазвичай, глядач – `viewer` – є особою або пристроєм. На початку програми локальний опорний простір та опорний простір глядача співпадають, та у процесі руху вони починають різнятись.

Метод `requestReferenceSpace` інтерфейсу `XRSession` повертає проміс, який стає екземпляром або `XRReferenceSpace`, або `XRBoundsReferenceSpace` відповідно до типу запитуваного опорного простору. Простір глядача запитується для того, щоб визначити відстані до об'єктів під час хіт-тесту від поточного положення глядача:

```
const viewerReferenceSpace =
    await session.requestReferenceSpace("viewer");
```

Таким чином, одночасно маємо два опорних простори – `local` та `viewer`.

Для запиту джерела хіт-тесту (`XRHitTestSource`) викликається асинхронна (через те, що повертає проміс) функція `requestHitTestSource`. Ця функція приймає словник `XRHitTestOptionsInit` з наступними параметрами ключ-значення:

- `space` (обов'язковий параметр) – опорний простір, який буде відстежуватися джерелом перевірки дотику: оскільки цей опорний простір оновлює своє місце знаходження для кожного кадру, `XRHitTestSource` буде рухатися разом з ним;
- `entityTypes` (необов'язковий параметр) – масив, що визначає типи сущностей, які будуть використані для створення джерела хіт-тесту. Якщо тип сущності не вказано, то за замовчуванням масив містить один елемент типу `plane`. Можливі типи:

- `point` – результати хіт-тесту обчислюються на основі виявлених характерних точок;
  - `plane` – результати хіт-тесту обчислюються на основі виявлених реальних площин;
  - `mesh` – результати хіт-тесту обчислюються на основі виявлених об'єктних сіток (віртуальних об'єктів).
- `offsetRay` (необов'язковий параметр) – об'єкт `XRRay` (промінь), який буде використано для виконання хіт-тесту: за замовчанням, промінь спрямований уперед, а координати його початку визначаються початком координат опорного простору.

Таким чином, `hitTestSource` є джерелом хіт-тесту для опорного простору `viewerReferenceSpace`:

```
const hitTestSource = await
  session.requestHitTestSource({space: viewerReferenceSpace});
```

Завершує обробник події `sessionstart` налаштування циклу анімації викликом `setAnimationLoop` – у WebXR вона має застосовуватись замість традиційної `requestAnimationFrame`. Безіменна функція, що встановлюється викликом `setAnimationLoop`, приймає два параметри – `timestamp` (поточний час) та `frame` (поточний кадр – інформація про фізичне середовище у момент часу `timestamp`):

```
renderer.setAnimationLoop((timestamp, frame) => {
```

Змінна `frame` є представником інтерфейсу `XRFrame` – якщо вона встановлена у `null`, інформація про фізичне середовище відсутня, тому подальша робота не є можливою:

```
if (!frame) return;
```

Метод `getHitTestResults` інтерфейсу `XRFrame` повертає масив `hitTestResults` об'єктів класу `XRHitTestResult`, що містить результати хіт-тесту для заданого джерела `hitTestSource`:

```
const hitTestResults = frame.getHitTestResults(hitTestSource);
```

Документація (<https://www.w3.org/TR/webxr-hit-test-1/>) не визнає, чи впорядковані результати хіт-тесту за відстанню так само, як результати рейткастингу, проте, згідно опису у <https://immersive-web.github.io/hit-test/hit-testing-explainer.html>, наразі це так, тому, якщо масив `hitTestResults` не порожній, його елемент з індексом 0 відповідає найближчому реальному об'єкту, до якого можна доторкнутись:

```
if (hitTestResults.length > 0) {  
    const hit = hitTestResults[0];
```

Метод `getReferenceSpace` повертає посилання на локальний опорний простір, встановлений для поточної сесії у модулі `ARButton`:

```
const referenceSpace = renderer.xr.getReferenceSpace();
```

Метод `getPose` інтерфейсу `XRFrame` повертає відносне положення та орієнтацію – позицію – одного простору `XRSpace` відносно іншого простору. За допомогою цього можна спостерігати за рухом об'єктів відносно одиного та фікованих локацій по всій сцені.

Таким чином, виклик `getPose` із параметром `referenceSpace` надає можливість визначити положення локального простору по відношенню до простору глядача:

```
const hitPose = hit.getPose(referenceSpace);
```

Наступний крок – зробити кільце видимим та розмістити його на поверхні, до якої виконано дотик:

```
reticle.visible = true;  
reticle.matrix.fromArray(hitPose.transform.matrix);
```

Якщо немає жодного реального об'єкту, до якого можна доторкнутись, кільце приховується:

```
} else {  
    reticle.visible = false;  
}
```

Завершується функція анімації викликом функції `render`:

```
    renderer.render(scene, camera);  
});
```

Для повноти додамо обробник парної події `sessionend`, що просто демонструє її перехоплення:

```
renderer.xr.addEventListener("sessionend", () => {  
    console.log("Сесію WebXR завершено");  
});  
  
initialize();  
});
```

## 6 Інтеграція машинного навчання

### 6.1 Спільний доступ до відео з камери

Для машинного навчання у Інтернеті найчастіше використовують TensorFlow (<https://www.tensorflow.org/>) – безкоштовну бібліотеку машинного навчання з відкритим вихідним кодом, розроблену компанією Google. На сьогоднішній день вона підтримує багато мов, включаючи основні – Python, Java, C++ – та підтримувані спільнотою: Haskell, C#, Julia, R, Ruby, Rust, Scala. Вона доступна на багатьох платформах, включаючи Linux, Window, Android, а також убудованих платформах – версія бібліотеки TensorFlow Lite призначена для роботи з моделями машинного навчання на мобільних пристроях, мікроконтроллерах, пристроях Інтернету речей тощо.

TensorFlow.js (<https://www.tensorflow.org/js>) – версія TensorFlow на JavaScript, що надає можливість розробляти та використовувати моделі, послуговуючись цією мовою, безпосередньо у браузері.

TensorFlow.js поставляється з великою кількістю попередньо навчених моделей, які можна одразу використовувати (<https://www.tensorflow.org/js/models>). Повний перелік моделей, доступних на поточний момент, подано за посиланням <https://github.com/tensorflow/tfjs-models> – багато із них є надзвичайно корисними і можуть стати гарним доповненням до AR-додатків. Якщо необхідна функціональність відсутня, можна створити та навчити власні моделі, або модифікувати наявні.

Розглянемо найпростіший приклад інтеграції моделі TensorFlow у AR-додаток з використанням бібліотеки MindAR. Для цього створимо наступний документ:

```
<html>
  <head>
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0">
    <script src=
"https://cdn.jsdelivr.net/npm/mind-ar/dist/mindar-image-three.prod.js"
      ></script>
```

```

<script>
    window.tf = window.MINDAR.IMAGE.tf;
</script>
<script src=
"https://unpkg.com/@tensorflow-models/handpose/dist/handpose.js">
</script>
<script src=".main.js" type="module"></script>
<style>
    html, body {position: relative; margin: 0; width: 100%;
                height: 100%; overflow: hidden}
</style>
</head>
<body>
</body>
</html>

```

Першим підключається бібліотека MindAR – точніше, та її частина, що відповідає за виявлення та відстеження зображень: `mindar-image-three.prod.js`. TensorFlow.js є її частиною, тому єдина необхідна дія – це створення властивості `tf` вікна `window` як копії `window.MINDAR.IMAGE.tf`. Якби бібліотека MindAR не використовувалась, то підключення TensorFlow.js виконувалося б у такий спосіб:

```

<script src=
"https://cdn.jsdelivr.net/npm/@tensorflow/tfjs/dist/tf.min.js"
></script>
<script src=
"https://unpkg.com/@tensorflow/tfjs-core/dist/tf-core.js"></script>
<script src=
"https://unpkg.com/@tensorflow/tfjs-converter/dist/tf-converter.js"
></script>

```

Моделі не є частиною TensorFlow.js, тому їх необхідно підключати окремо – так, як показано на прикладі моделі `handpose.js`, описаної за посиланням <https://github.com/tensorflow/tfjs-models/tree/>

`master/hand-pose-detection`. Ця модель використовується для визначення кисті руки та її складових.

Файл `main.js` містить основний код програми, на початку якої визначається асинхронна функція `start`, що виконується після того, як завершується завантаження документу HTML:

```
const THREE = window.MINDAR.IMAGE.THREE;

document.addEventListener("DOMContentLoaded", () => {
    const start = async() => {
        // вміст функції start
    }
    start();
});
```

Першим створюється примірник AR-рушія (об'єкт `mindarThree`), що ідентифікуватиме зображення, задане набором опорних точок `altabor.mind`, у відеопотоці в тілі документа:

```
const mindarThree = new window.MINDAR.IMAGE.MindARThree({
    container: document.body, imageTargetSrc: "altabor.mind",
});
```

Об'єкт `mindarThree` містить необхідні для подальшої роботи властивості:

```
const {renderer, scene, camera} = mindarThree;
```

`plane` є квадратною напівпрозорою площиною ціанового кольору розміром  $1 \times 1$  (у одиницях ширини зображення):

```
const geometry = new THREE.PlaneGeometry(1, 1);
const material = new THREE.MeshBasicMaterial({
    color: 0x00ffff, transparent: true, opacity: 0.5
});
const plane = new THREE.Mesh(geometry, material);
```

`anchor` – якір, пов’язаний із відстежуваним зображенням:

```
const anchor = mindarThree.addAnchor(0);
```

Додавання площини до групи об’єктів, пов’язаної з якорем, забезпечує її відображення, коли бібліотека знаходить відстежуване зображення у відеопотоці:

```
anchor.group.add(plane);
```

Наступний крок – завантаження моделі `handpose` з TensorFlow Hub (<https://tfhub.dev/>) – переглянувши цей репозитарій моделей, можна побачити, що вони займають чималий обсяг, тому метод `load`, що їх завантажує, викликається як асинхронна функція:

```
const model = await handpose.load();
```

Наступний крок – запуск AR-рушія:

```
await mindarThree.start();
```

Цикл анімації розділимо на дві частини: перша буде включати лише базовий рендеринг та налаштовується викликом `setAnimationLoop`:

```
renderer.setAnimationLoop(() => {
  renderer.render(scene, camera);
});
```

Модель `handpose` опрацьовує окремі кадри, які беруться з відеопотоку `video`. Це достатньо обчислювально ємна процедура, тому, ураховуючи, що, доки велика точність ідентифікації рук непотрібна, можна спробувати виявляти їх не у кожному кадрі, а, наприклад, у кожному десятому. Для цього використовується лічильник кадрів `frameCount`:

```
const video = mindarThree.video;
```

```
let frameCount = 1;
```

Функція `detect` утворює другий, більш змістовний, цикл анімації:

```
const detect = async () => {
```

Для кожного десятого кадру виконується виклик методу `estimateHands` завантаженої моделі, якому передається кадрі `video`:

```
if (frameCount % 10 == 0) {  
    const predictions = await model.estimateHands(video);
```

Метод повертає масив `predictions`, який містить відомості про детектовані у кадрі зображення рук, тому ненульовий розмір масиву – ознака того, що у кадрі була рука:

```
if (predictions.length > 0) {
```

На жаль, дійсно поки що лише одна рука – поточна версія моделі (станом на 16.11.2022) дає можливість визначити лише одну руку, тому звернемось до першого елементу масиву `predictions` для отримання даних про неї. Серед багатьох детектованих властивостей, що містить цей елемент, є, наприклад, `boundingBox`, елементами якого є `topLeft` та `bottomRight` – відповідно лівий верхній та правий нижній кути прямокутника, що обмежує руку. Відповідно, `w` та `h` є відносними розмірами руки – вони будуть дорівнювати 1 лише у випадку, коли рука займатиме весь кадр:

```
const x = predictions[0].boundingBox.topLeft[0];  
const y = predictions[0].boundingBox.topLeft[1];  
const x1 = predictions[0].boundingBox.bottomRight[0];  
const y1 = predictions[0].boundingBox.bottomRight[1];  
const w = (x1 - x) / window.innerWidth,  
        h = (y1 - y) / window.innerHeight;
```

Якщо рука є видимою, зробимо видимим її площину `plane`, масштабувавши її за відносними розмірами руки:

```
plane.visible = true;  
plane.scale.set(w, h, 1);
```

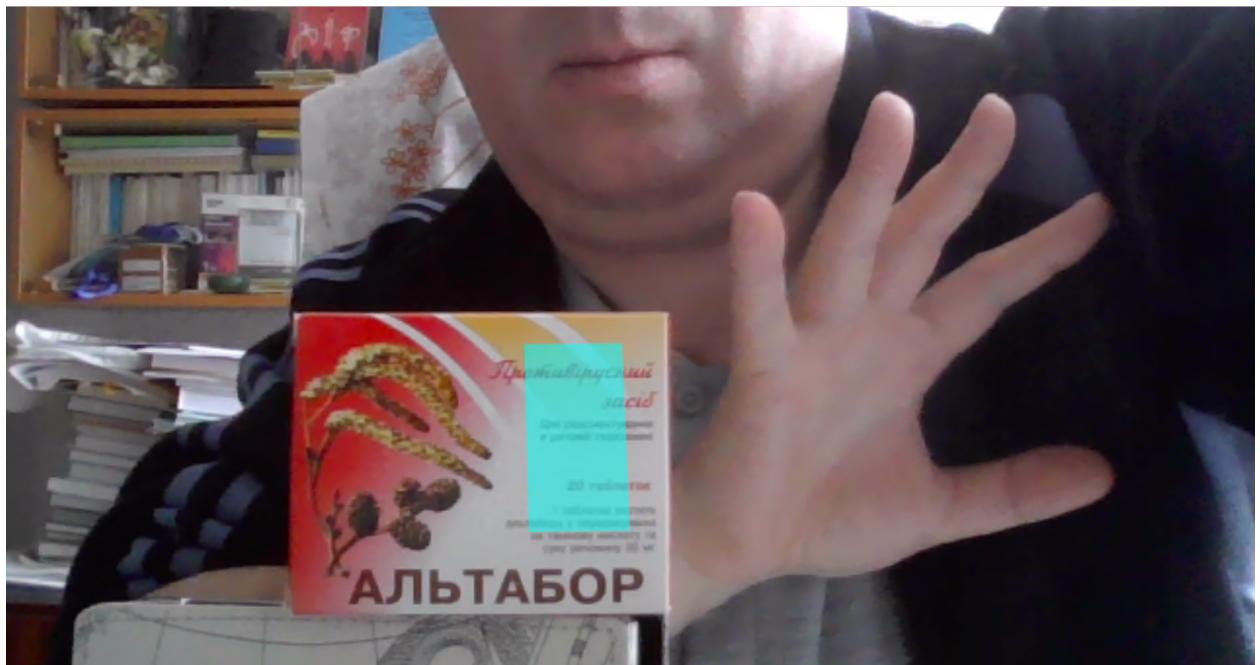


Рис. 6.1. Жестове управління розміром та положенням віртуального об'єкта.

Положення площини на детектованому зображені встановимо так, щоб воно відображало положення обмежувального прямокутника руки у кадрі (рис. 6.1):

```
plane.position.set(x/window.innerWidth,  
y/window.innerHeight, 0);  
}
```

Якщо руки у кадрі немає, площину не показуватимемо:

```
else  
    plane.visible = false;  
}
```

Наприкінці функції `detect` викликом `requestAnimationFrame` повідомляємо браузеру, що вона буде йому необхідно оновити положення саме за допомогою цієї функції, та збільшуємо на 1 лічильник кадрів:

```
frameCount++;  
window.requestAnimationFrame(detect);  
}
```

Остання дія у функції `start` – запуск другого циклу анімації викликом `requestAnimationFrame`:

```
window.requestAnimationFrame(detect);
```

Створений ефект достатньо простий, проте він надає уявлення про те, як використовувати моделі машинного навчання у AR-додатках.

## 7 Інші технології WebAR

- 7.1 Вступ до A-Frame
- 7.2 A-Frame та WebXR
- 7.3 Вступ до model-viewer
- 7.4 Вступ до комерційних SDK

## **Висновки**

# 8 Лабораторні роботи

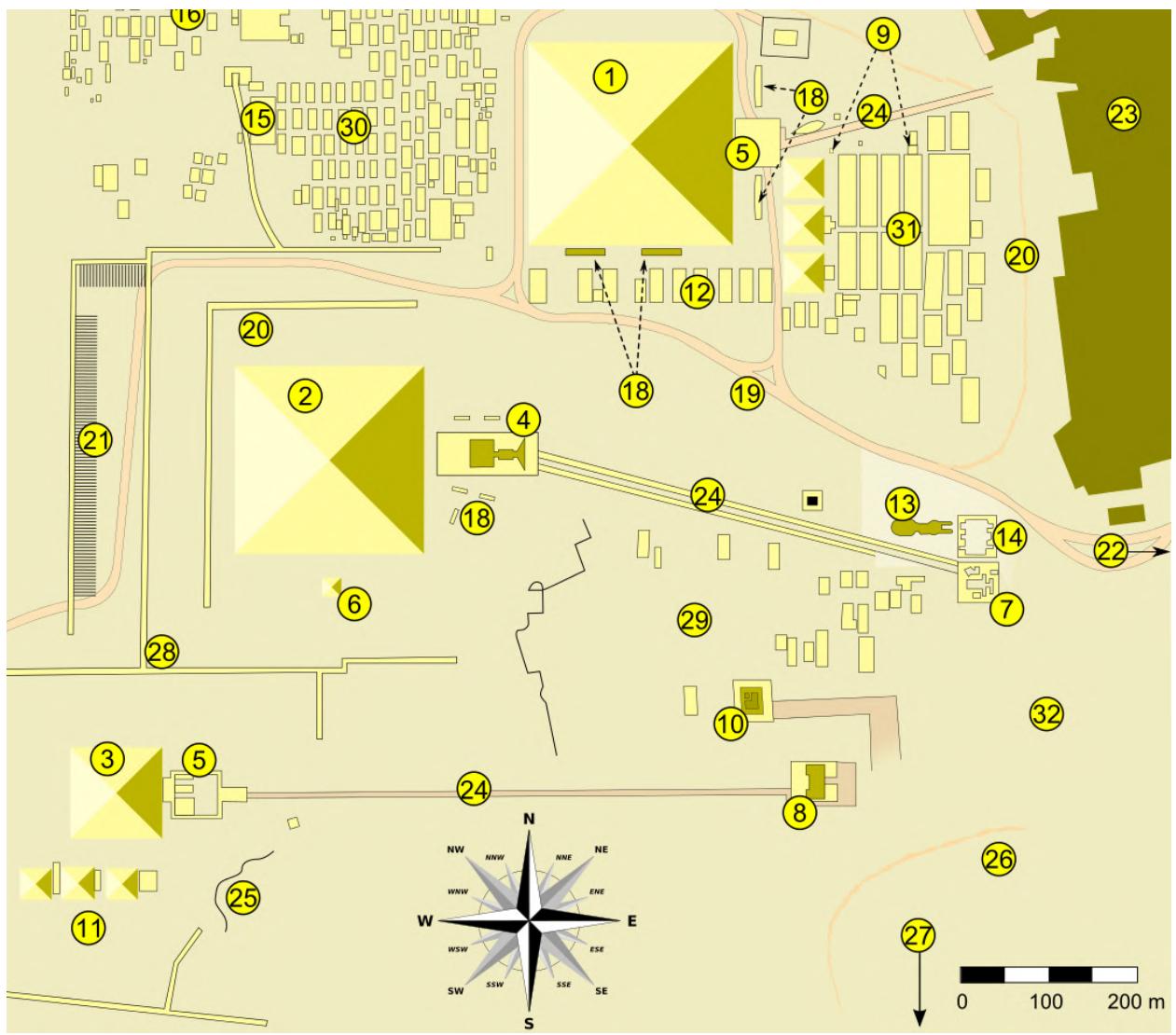
## 8.1 Перша

### Варіант 0

1. Створіть масштабовану модель Земля-Місяць. Землю і Місяць вважайте сферами. Радіус Землі – 6 400 км, Місяця – 1 740 км. Зверніть увагу на параметр сфери segments-height. Поекспериментуйте з різними його значеннями. Потрібні текстури можна знайти в репозитарії A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/space>
2. Анімуйте модель.Період обертання Землі навколо своєї осі – 1 доба, період обертання Місяця навколо Землі – 28 діб. Масштаб часу – 1:17 000.
3. Додайте джерело світла на відстані 30 000 км.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати фотографію Землі, знану як “Blue Marble” (її можна знайти, наприклад, у статті Wikipedia [https://en.wikipedia.org/wiki/The\\_Blue\\_Marble](https://en.wikipedia.org/wiki/The_Blue_Marble)).

### Варіант 1

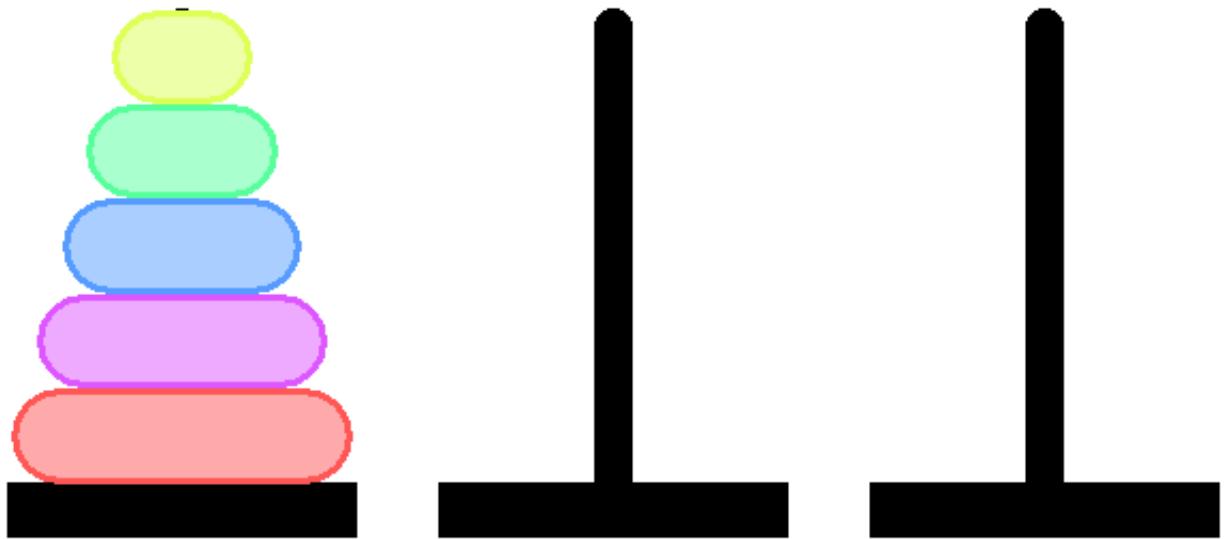
1. Створіть модель храмового комплексу Гізи (зобразіть три найбільші піраміди: Хеопса (1), Хефрена (2) та Мікеріна (3)). Задля того, щоб правильно задати розміри, можете скористатися схемою. Висота піраміди Хеопса – 145 м (довжина бокової грані – 230 м); піраміди Хефрена – 135 м (210 м), піраміди Мікеріна – 65 м (100 м).
2. Додайте до пірамід “цегляну” текстуру (можете знайти їх в колекції A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/bricks>). Верхівки пірамід “пофарбуйте” в золотавий колір.
3. Додайте до моделі камеру, яка б “наїжджає” на центральну піраміду.



4. “Прив’яжіть” створену сцену до маркеру дополненої реальності. Як маркер можете використати прапор Гізи (його можна знайти, наприклад, у колекції Wikipedia: [https://commons.wikimedia.org/wiki/File:Governorat\\_de\\_Gizeh.png](https://commons.wikimedia.org/wiki/File:Governorat_de_Gizeh.png)).

## Варіант 2

1. Побудуйте модель Ханойських веж з трьома стрижнями та сімома кільцями різних розмірів та всіх кольорів веселки.
2. Анімуйте поступове переміщення парних кілець на середній стрижень, а непарних – на правий.
3. “Прив’яжіть” створену сцену до маркеру дополненої реальності. Як маркер можете використати герб Ханоя (його можна знайти, наприклад, у колекції Wikipedia: <https://commons.wikimedia.org/wiki/>



File:Emblem\_of\_Hanoi.svg).

### Варіант 3

1. Створіть масштабовану модель Сатурну з його системою кілець D. Радіус Сатурна – 60 000 км. Внутрішній радіус кільця D – 74 500 км, зовнішній радіус кільця A – 137 000 км. Зверніть увагу на параметр сфери segments-height. Поекспериментуйте з різними його значеннями. Потрібні текстири можна завантажити з порталу Solar Textures: <https://www.solarsystemscope.com/textures>
2. Анімуйте модель. Сатурн обертається навколо своєї осі. При цьому нахил осі складає  $27^\circ$ , а кільця лежать в екваторіальній площині Сатурну.
3. Додайте джерело світла на відстані 500 000 км.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати фрагмент картини Рубенса “Сатурн”, який можна знайти, наприклад, у колекції Wikipedia: <https://commons.wikimedia.org/wiki/File:Rubens-Saturno-detalle.jpg>.

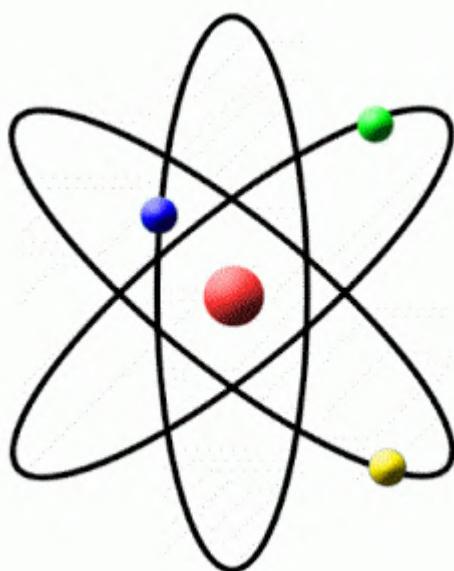
### Варіант 4

1. Створіть схематичний макет будівлі Криворізької міської ради. Для цього можете скористатися планом будівлі <https://www.google.com>.

2. Додайте до будівлі “цегляну” текстуру (можете знайти їх в колекції A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/bricks>).
3. Додайте до моделі камеру, яка б “від’їжджала” б від будівлі.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати герб Кривого Рогу (його можна знайти, наприклад, у статті Wikipedia [https://uk.wikipedia.org/wiki/%D0%93%D0%B5%D1%80%D0%B1\\_%D0%9A%D1%80%D0%B8%D0%B2%D0%BE%D0%B3%D0%BE\\_%D0%A0%D0%BE%D0%B3%D1%83](https://uk.wikipedia.org/wiki/%D0%93%D0%B5%D1%80%D0%B1_%D0%9A%D1%80%D0%B8%D0%B2%D0%BE%D0%B3%D0%BE_%D0%A0%D0%BE%D0%B3%D1%83)).

### Варіант 5

1. Побудуйте планетарну модель атома літію. Орбіти електронів вважайте коловими; площині орбіт нахилені одна відносно одної; радіуси орбіт різні. Електрони вважайте кулями радіуса  $10^{-17}$  м, а ядро – кулею радіуса  $10^{-15}$  м, радіус першої орбіти –  $10^{-14}$  м, другої –  $5 \cdot 10^{-14}$  м, третьої –  $10^{-13}$  м. Зверніть увагу на параметр сфери segments-height. Поекспериментуйте з різними його значеннями.



2. Анімуйте модель. Вкажіть різні періоди обертання різних атомів.

3. Додайте до ядра якусь текстуру з колекції A-Frame <https://github.com/aframevr/sample-assets/tree/master/assets/images/noise>.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати герб Резерфорда (його можна знайти, наприклад, у колекції Wikipedia [https://commons.wikimedia.org/wiki/File:Ernest\\_Rutherford\\_Arms.svg](https://commons.wikimedia.org/wiki/File:Ernest_Rutherford_Arms.svg)).

## Варіант 6

1. Створіть модель гумової кульки, що падає на похилу площину. Зверніть увагу на параметр сфери segments-height. Поекспериментуйте з різними його значеннями. Потрібні текстири можна знайти в репозитарії A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/noise>, <https://github.com/aframevr/sample-assets/tree/master/assets/images/wood>
2. Анімуйте модель. Удари вважайте абсолютно пружними.
3. Додайте камеру, яка рухалася б вздовж похилої площини, супроводжуючи кульку.
4. Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати емблему NBA (її можна знайти, наприклад, у колекції Wikipedia [https://en.wikipedia.org/wiki/File:National\\_Basketball\\_Association\\_logo.svg](https://en.wikipedia.org/wiki/File:National_Basketball_Association_logo.svg)).

## Варіант 7

1. Створіть модель циліндра, що скочується похилою площиною. Зобразіть вектори всіх сил, що діють на циліндр. Потрібні текстири можна знайти в репозитарії A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/noise>, <https://github.com/aframevr/sample-assets/tree/master/assets/images/wood>.
2. Анімуйте модель.

3. Додайте джерело світла.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати емблему гурту The Rolling Stones (її можна знайти, наприклад, у колекції Wikipedia [https://en.wikipedia.org/wiki/File:The\\_Rolling\\_Stones\\_Tongue\\_Logo.png](https://en.wikipedia.org/wiki/File:The_Rolling_Stones_Tongue_Logo.png)).

## Варіант 8

1. Створіть “модель” Великого адронного колайдера: два протона-кульки, що летять всередині прозорого тору.
2. Анімуйте модель: частинки мають рухатися всередині тора назустріч одна одній, доки не зіткнуться.
3. Додайте джерело світла.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати логотип експерименту ATLAS (їого можна знайти, наприклад, у колекції Wikipedia <http://cds.cern.ch/record/1170735/files/ATLAS-Logo-Standard.png?subformat=icon-1440>).

## Варіант 9

1. Схематично намалюйте фортецю, використовуючи циліндри, конуси, прямокутні паралелепіпеди, сфери. “Підніміть” над фортецею прапор із надписом “Thank You Mario But Our Princess Is in Another Castle”.
2. Додайте до фортеці “цегляну” текстуру (можете знайти їх в колекції A-Frame: <https://github.com/aframevr/sample-assets/tree/master/assets/images/bricks>).
3. Анімуйте модель: нехай прапор спочатку піднімається по флагштоку, а потім розвивається на вітру.
4. “Прив’яжіть” створену сцену до маркеру доповненої реальності. Як маркер можете використати якесь із зображень Mario або його ем-

блему (їх можна знайти, наприклад, у статті Wikipedia <https://en.wikipedia.org/wiki/Mario>).

## 9 Віртуальна хімічна лабораторія у доповнений реальності: від ідеї до реалізації

### 9.1 Ідея

*Загальна:* створити віртуальну хімічну лабораторію, дії в якій над віртуальними об'єктами, заданими зображеннями або моделями реальних об'єктів, приводять до відтворення процесу їх еталонної взаємодії.

*Частинна:* для набору реактивів, необхідних для виконання лабораторної роботи “Якісні реакції хлорид-, бромід- та йодид-йонів”, надати учням можливість випробувати всі варіантів їх поєднання.

Для реалізації даної ідеї пропонується кожний із реактивів співставити із маркером доповненої реальності. Для “зливання” реактивів два маркери необхідно наблизити один до одного й, по досягненню певної відстані, відобразити відеозапис реального експерименту.

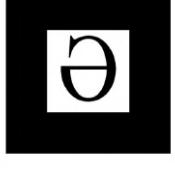
### 9.2 Підготовка маркерів, зображень та відео

Для підготовки маркерів необхідно обрати засіб компіляції зображень та видлення їх опорних точок, що відповідає застосованому рушію доповненої реальності. Так, при виборі в якості рушія AR.js таким компілятором буде “AR.js Marker Training” (<https://ar-js-org.github.io/AR.js/three.js/examples/marker-training/examples/generator.html>), що генерує кольорові квадратні маркери роздільною здатністю  $16 \times 16$  у форматі .patt: 3 кольорові площини (червона, зелена та синя) и 4 орієнтаціях (кути повороту  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$  та  $270^\circ$ ).

Згенеровані маркери пронумеровані від 0 до 9 та співставлені із назвами і зображеннями реактивів (табл. 9.1).

Таблиця 9.1

Таблиця відповідності реактивів, маркерів та зображенень.

№	Реактив	Маркер	Зображення
0	AgNO <sub>3</sub>	pattern-0.patt 	AgNO3.jpg 
1	Pb(NO <sub>3</sub> ) <sub>2</sub>	pattern-1.patt 	Pb(NO <sub>3</sub> ) <sub>2</sub> .jpg 
2	HCl	pattern-2.patt 	HCl.jpg 

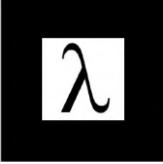
Продовження табл. 9.1

<b>№</b>	<b>Реактив</b>	<b>Маркер</b>	<b>Зображення</b>
3	KCl	pattern-3.patt 	КCl.jpg 
4	NaCl	pattern-4.patt 	NaCl.jpg 
5	NH <sub>4</sub> Cl	pattern-5.patt 	NH <sub>4</sub> Cl.jpg 

Продовження табл. 9.1

<b>№</b>	<b>Реактив</b>	<b>Маркер</b>	<b>Зображення</b>
6	KBr	pattern-6.patt 	KBr.jpg 
7	NH <sub>4</sub> Br	pattern-7.patt 	NH4Br.jpg 
8	KI	pattern-8.patt 	KI.jpg 

Продовження табл. 9.1

№	Реактив	Маркер	Зображення
9	NH <sub>4</sub> I	pattern-9.patt 	NH4I.jpg 

Попарне застосування 10 маркерів без урахування порядку комбінування дає такі 45 комбінацій:

0-1, 0-2, 0-3, 0-4, 0-5, 0-6, 0-7, 0-8, 0-9

1-2, 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, 1-9

2-3, 2-4, 2-5, 2-6, 2-7, 2-8, 2-9

3-4, 3-5, 3-6, 3-7, 3-8, 3-9

4-5, 4-6, 4-7, 4-8, 4-9

5-6, 5-7, 5-8, 5-9

6-7, 6-8, 6-9

7-8, 7-9

8-9

Далеко не всіх з них мають сенс, тому відеозапис було виконано лише для тих комбінацій реактивів, в яких наявні ознаки хімічної реакції різняться (табл. 9.2).

Для випадків, коли реакція не відбувається, були додатково записані файли 09.mp4, 14.mp4, 1.1.mp4, 1.2.mp4, 1.3.mp4, 1.4.mp4, 1.5.mp4.

Створені файли маркерів, зображень та відео були завантажені у репозитарій GitHub за посиланням <https://github.com/Lefoxi/ARdip> до каталогів markers, images та video відповідно. Додатково для зручності використання зображення маркерів були зібрані у файлі markers.pdf (<https://github.com/Lefoxi/ARdip/blob/main/markers.pdf>).

Таблиця 9.2

Розподіл відеозаписів по маркерах.

<b>№</b>	<b>Реактив</b>	<b>№</b>	<b>Реактив</b>
0	AgNO <sub>3</sub>	1	Pb(NO <sub>3</sub> ) <sub>2</sub>
2	HCl	16.mp4	08.mp4
3	KCl	10.mp4	11.mp4
4	NaCl	10.mp4	13.mp4
5	NH <sub>4</sub> Cl	17.mp4	07.mp4
6	KBr	06.mp4	05.mp4
7	NH <sub>4</sub> Br	03.mp4	04.mp4
8	KI	02.mp4	01.mp4
9	NH <sub>4</sub> I	02.mp4	01.mp4

## 9.3 Програмна реалізація

### 9.3.1 Початкова сторінка

Початкова сторінка – файл index.html (додаток А), тіло якого містить два гіперпосилання:

- ar.html – власне код віртуальної лабораторії;
- markers.pdf – файл PDF із зображеннями маркерів для друку.

### 9.3.2 Реалізація віртуальної лабораторії

Для реалізації лабораторії було обрано бібліотеку A-Frame для створення віртуальних об'єктів та бібліотеку AR.js для їх зв'язування зі створеними маркерами.

Загальна структура файлу ar.html:

```
<html>
  <head>
    <!-- налаштування заголовку вікна -->
    <title>
      Віртуальна хімічна лабораторія у доповненій реальності
    </title>
  </head>
  <body>
    <script src="https://aframe.io/releases/0.11.0/aframe.min.js"></script>
    <script src="https://cdn.AR.js/v1/AR.js"></script>
    <script>
      // Your custom code here
    </script>
  </body>
</html>
```

```

<!-- підключення бібліотеки A-Frame -->
<script
    src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
<!-- підключення бібліотеки AR.js для роботи з A-Frame -->
<script src=
"https://raw.githack.com/AR-js-org/AR.js/master/aframe/build/aframe-ar
"></script>
<script>
    <!-- основна програма -->
</script>
</head>

<body>
    <!--Створення сцени A-Frame для роботи у доповненій реальності-->
    <a-scene vr-mode-ui="enabled: false;">
        renderer="logarithmicDepthBuffer: true;" embedded
        arjs="trackingMethod: best; sourceType: webcam; debugUIEnabled: false;
        >
            <!-- Наповнення сцени A-Frame -->
        </a-scene>
    </body>
</html>

```

Сцену A-Frame складають 4 блоки:

- 1) <a-assets>...<a-assets> – визначення відеотекстур;
- 2) <a-marker>...</a-marker> – визначення маркерів та прив'язка до них зображень і відео;
- 3) <a-entity camera></a-entity> – розміщення на сцені камери;
- 4) <a-entity run></a-entity> – звернення до компоненту **run**, визначеного у основній програмі.

Блок визначення відеотекстур складають записи виду:

```
<video id="video02" preload="auto" muted poster="images/starting.jpg"  
loop="false" src="video/16.mp4"></video>
```

- **id** – ідентифікатор відео, що складається з імені `video` та номерів маркерів, що суміщуються (згідно табл. 9.2). Якщо одне й те саме відео застосовується для декількох комбінацій маркерів, ідентифікатор набуває вигляду `video03-04`. Для випадків, коли реакція не відбувається, передбачені відеотекстури `video00-1` ... `video00-7`;
- встановлення `preload` у `auto` визначає необхідність завантаження відео до першого звернення до нього – це збільшує початковий час завантаження програми, проте зменшує час, необхідний для появи відео і убезпечує від зупинок відео через нестабільне мережне з'єднання;
- `muted` – вказує на необхідність вимкнення звуку, якщо звукова доріжка наявна у відео;
- `poster` містить посилання на файл зображення, що з'являється, якщо звернення до відео відбулось до того, як воно повністю завантажилось;
- `src` – посилання на відеофайл.

Блок визначення маркерів складають 9 записів виду:

```
<a-marker type="pattern" url="markers/pattern-0.patt" id="M0"  
registerevents>  
<!-- Визначення зображення --&gt;<br/><!-- Визначення відео --&gt;<br/></a-marker>
```

Кожен маркер визначається номером від 0 до 9 (згідно табл. 9.1), який входить до складу імені маркера (`pattern-0.patt` – файл опорних точок маркеру 0) та його ідентифікатора `id` (`M0` – ідентифікатора маркеру 0). До кожного маркера застосовується компонент `registerevents`, визначеній в основній програмі.

Визначення зображення, що накладається на маркер після його виявлення, виконується командою виду:

```
<a-plane src="images/AgN03.jpg" rotation="-90 90 0" id="draw0">  
</a-plane>
```

`a-plane` визначає текстуровану площину. Параметр `src` містить посилання на файл текстури – зображення реактиву згідно табл. 9.1. Через те, що зображення збережено у альбомному розташуванні, а демонструється у портретному, виконується його поворот на кут  $90^\circ$  навколо вісі  $OY$ . Поворот на  $270^\circ$  ( $-90^\circ$ ) навколо вісі  $OX$  необхідний через те, що координатний простір A-Frame повернутий на відповідний кут відносно координатного простору камери. `id` – ідентифікатор площини із зображенням, що складається зі слова `draw` та номера маркера.

До кожного маркера прив'язується від 9 (маркер 0) до 0 (маркер 9) площин із відео згідно визначеної вище схеми комбінацій маркерів:

```
<a-entity  
  material="shader: flat; src: #video00-1"  
  geometry="primitive: plane; width: 0.90; height: 1.60;"  
  position="0 0 0" scale="1.5 1.5 1.5" rotation="-90 0 0"  
  id="drawX01" visible="false">  
</a-entity>
```

Співвідношення сторін площини задається її шириною `width` та висотою `height` (вказуються відносно розміру маркера, що приймається за 1). Властивість `src` матеріалу площини містить посилання на раніше визначену та завантажену відеотекстуру. Параметр `position` визначає координати центру площини ( $0\ 0\ 0$  – співпадіння з координатами центру маркера), `scale` – півторократне масштабування ( $1.5\ 1.5\ 1.5$ ) відеотекстури порівняно із визначеними шириною та висотою, `rotation` визначає поворот відповідний кут (аналогічно до повороту зображення). Ідентифікатор об'єкту `id` складається зі слова `drawX` та номерів маркерів, що зближаються, впорядкованих за зростанням.

Основна програма складається з 3 блоків:

- 1) визначення змінних:

- `howmuch` – емпірично дібране значення граничної відстані між маркерами, менше за яку вважатимемо, що має розпочатись реакція (за замовчанням 1.4);
  - `isReaction` – прапорець, що вказує на те, що реакція розпочалась: якщо він встановлений, демонструється відповідний відеозапис реакції (початкове значення `false`);
  - `distance` – поточна відстань між парою маркерів, що зближуються (початкове значення `howmuch+1`);
  - `markerVisible` – масив станів видимості маркерів: якщо маркер видимий, значення елементу масиву з відповідним маркером номером встановлюється у `true` (початкове значення для кожного маркера `false`);
  - `M` – масив для збереження маркерів (визначених у документі HTML об'єктів `a-marker`);
  - `d` – масив для збереження зображень, що накладаються на маркери (визначених у документі HTML об'єктів `a-plane`, ідентифіковані як `drawN`, де `N` – номер маркера);
  - `X` – масив для збереження відео, що накладаються на маркери (визначених у документі HTML об'єктів `a-plane`, ідентифіковані як `drawXAB`, де `A, B` – номери маркерів, що зближуються);
  - `r` – масив координат центрів маркерів, значення якого використовуються для вимірювання відстаней;
  - `isVideoPlay` – прапорець, що вказує на те, що відтворюється відеозапис реакції (початкове значення `false`);
- 2) реєстрація компоненту `registerevents`;
- 3) реєстрація компоненту `run`.

Компонент `registerevents` призначений для відстеження двох подій – виявлення (`markerFound`) та втрати (`markerLost`) маркера. Після реєстрації метод `init` компоненту застосовується до усіх маркерів, до яких був доданий компонент `registerevents`, з метою визначення номеру маркера `index` та встановлення при виявленні чи зняття при втраті пропорція

його видимості у масиві `markerVisible`. При виявленні маркеру пов'язане із ним зображення `d[index]` робиться видимим, а при втраті – невидимим встановлення у відповідне значення властивості `visible`:

```
AFRAME.registerComponent("registerevents", {
  init: function () {
    let marker = this.el;
    marker.addEventListener('markerFound', function() {
      index = parseInt(marker.id[1])
      markerVisible[ index ] = true;
      d[index] = document.querySelector("#draw"+index);
      if(d[index]!=null)
        d[index].setAttribute("visible", "true");
    });
    marker.addEventListener('markerLost', function() {
      index = parseInt(marker.id[1])
      markerVisible[ index ] = false;
      if(d[index]!=null)
        d[index].setAttribute("visible", "false");
    });
  }
});
```

Компонент `run` не відповідає жодному видимому елементу – він виконує загальне управління сценою, для чого реєструється як одноразово виконуваний метод `init`, так і виконуваний за таймером (бажано для кожного кадру) метод `tick`:

```
AFRAME.registerComponent("run", {
  init: function() {
    // ініціалізація компоненту
  },

  tick: function (time, deltaTime) {
    // метод, що викликається за таймером
  }
})
```

```
});
```

У методі `init` заповнюються визначені раніше масиви: `M` – посиланнями на маркери, `d` – посиланнями на зображення реактивів (відображаються для маркерів, що є видимими), `X` – посиланнями на прив'язане до маркерів відео, та `p` – нульовими координатними векторами:

```
for (let i = 0; i < 10 ; i++) {  
    M[i] = document.querySelector("#M"+i);  
    d[i] = document.querySelector("#draw"+i);  
    p[i] = new THREE.Vector3();  
    if (markerVisible [i])  
        d[i].setAttribute("visible", "true");  
    for (let j = 0; j < 10 ; j++)  
        if(i!=j)  
            X[i][j] = document.querySelector("#drawX"+i+"_"+j);  
}
```

Метод `tick` постійно відслідковує видимість та взаємне розташування маркерів:

- 1) до масиву `visible` заносимо номери маркерів, що у поточний момент видимі – для цього аналізуємо зміст масиву `markerVisible`, встановлений компонентом `registerevents`:

```
let visible = [] ;  
for (let i=0;i<10;i++)  
    if (markerVisible[i])  
        visible.push(i);
```

- 2) будемо вважати, що для реакції необхідні рівно 2 реактиви, тому випадки, коли видимими є менше двох маркерів (або більше двох), ігноруватимемо – якщо демонструвалось відео, робимо його невидимим та встановлюємо пропорець `isVideoPlay` у `false`:

```

if(visible.length!=2) {
    for (let i = 0; i < 10 ; i++)
        for (let j = 0; j < 10 ; j++)
            if(i!=j && X[i][j]!=null)
                X[i][j].setAttribute("visible", "false");
    isVideoPlay = false;
    return;
}

```

- 3) визначаємо номери двох видимих маркерів – `marker1` та `marker2`:

```

marker1 = visible[0];
marker2 = visible[1];

```

- 4) визначаємо координати маркерів та зберігаємо їх у відповідних елементах координатного масиву `p`:

```

M[marker1].object3D.getWorldPosition(p[marker1]);
M[marker2].object3D.getWorldPosition(p[marker2]);

```

- 5) знаходимо відстань між маркерами:

```

distance = p[marker1].distanceTo( p[marker2] );

```

- 6) у пропорці `isReaction` встановлюємо, чи достатньо близькі маркери для перебігу реакції:

```

isReaction=(distance <= howmuch);

```

- 7) пропорець `isReaction` не вказує, чи йшла реакція до поточного моменту – визначити це можна, проаналізувавши, чи вже програвалось відео: якщо ні, знімаємо з обох маркерів зображення, робимо видимим відповідне відео та розпочинаємо його програвати із самого початку:

```

if(isReaction) {
    if(!isVideoPlay) {

```

```

        if(d[marker1] !=null)
            d[marker1].setAttribute("visible", "false");
        if(d[marker2] !=null)
            d[marker2].setAttribute("visible", "false");
        X[marker1][marker2].setAttribute("visible", "true");
        var id="#"+X[marker1][marker2]
            .getAttribute("material").src.getAttribute("id");
        var video=document.querySelector(id);
        video.currentTime=0;
        video.play();
        isVideoPlay = true;
    }
}

```

- 8) якщо маркери віддались на відстань, що відповідає припиненню реакцію, призупиняємо відео, робимо його невидимим та знову накладаємо зображення реактивів на обидва маркери:

```

else {
    if(d[marker1] !=null)
        d[marker1].setAttribute("visible", "true");
    if(d[marker2] !=null)
        d[marker2].setAttribute("visible", "true");
    X[marker1][marker2].setAttribute("visible", "false");
    var id="#"+X[marker1][marker2]
        .getAttribute("material").src.getAttribute("id");
    var video=document.querySelector(id);
    video.currentTime=0;
    video.pause();
    isVideoPlay = false;
}

```

### 9.3.3 Розгортання програмного забезпечення

Ураховуючи, що вихідні тексти програмного забезпечення було розміщені у репозитарії GitHub, його розгортання було виконано на

сторінках GitHub (GitHub Pages). Для цього у налаштуваннях репозитарію (Settings – Pages) була обрана головна гілка репозитарію (main) та його кореневий каталог, що містить початкову сторінку (файл index.html), та виконано збереження (Save). Після цього виконується процедура розгортання програмного забезпечення (публікація сайт) за посиланням <https://lefoxi.github.io/ARdip/>. Наявність на опублікованому сайті файлу index.html надає можливість автоматично перейти до початкової сторінки після переходу за вказаним вище посиланням.

Для оновлення розгорнутого програмного забезпечення достатньо виконати завантажити змінені файли до репозитарію.

### 9.3.4 Тестування віртуальної лабораторії

... (*далі – самi*)

## A Файл index.html

```
<html>
  <head>
    <style>
      button {
        color: lightgreen; background: #0000aa; height: 150px;
        width: 450px; font-size: 200%; border: 2px solid powderblue;
        padding: 30px; position: fixed; padding: 1vh;
        bottom: 1vh; left: 50%; transform: translate(-50%, -50%);
        top: 50%;
      }
      #markers {
        font-size: 138%; border: 2px solid powderblue; padding: 30px;
        position: fixed; bottom: 1vh; left: 50%;
        transform: translateX(-50%);
      }
    </style>
  </head>
  <body>
```

```
<a href="ar.html" target="_blank"><button>Перейти до віртуальної  
лабораторії</button></a><p>  
<a href="markers.pdf" target="_blank" id="markers">Натисність для  
завантаження набору маркерів - роздрукуйте та розріжте їх. У  
віртуальній лабораторії покажіть їх камері, ѿ, після появи на  
маркері зображення, спробуйте їх зблизити, але без накладання.</a>  
</body>  
</html>
```

## Б Файл ar.html

Тут слід розмістити код відповідного файлу.