

## **Розробка програмного забезпечення із доповненою реальністю з використанням JavaScript**

*Мета* – навчитися створювати програми із доповненою реальністю, орієнтовані на традиційні та мобільні веб-браузери, за допомогою бібліотек A-Frame, Three.js, AR.js та ін.

*Пререквізити* – успішне опанування курсу «Web-програмування».

*Обладнання* – стаціонарний або мобільний комп’ютерний засіб, обладнаний зовнішньою або вбудованою камерою.

*Програмне забезпечення* – веб-браузер із підтримкою WebGL, текстовий редактор.

*Анотація.* Ідея доповненої реальності (Augmented Reality – AR) полягає у комбінуванні комп’ютерно-генерованих моделей з об’єктами реального світу. AR.js надає набір засобів для ефективної розробки AR-програм за допомогою веб-технологій, таких як JavaScript і HTML. Вступ містить основ AR та короткий огляд усіх бібліотек та засобів розробки. Робота розпочинається з налаштування базової структури AR-програми з використанням бібліотеки AR.js. Далі виконується налаштування камери для локалізації маркерів AR. Після цього виконується накладання 3D-контенту поверх маркерів за допомогою бібліотеки Three.js з використанням реалістичного освітлення та тіней до 3D-об’єктів з різними елементами керування, що надаються Three.js для навігації по 3D-сцені. Пропонується кілька проектів для закріplення набутих навичок створення AR-програм за допомогою A-Frame, Three.js, AR.js та інших бібліотек.

# 1 ТЕХНОЛОГІЇ ДОПОВНЕНОЇ РЕАЛЬНОСТІ

## 1.1. Вступ

Програмувати доповнену реальність – інноваційно (модно, цікаво, корисно та ін.) останні 60 років, і вибір JavaScript в якості мови програмування – виключно данина часу. Але вибір мови визначає й вибір засобів розробки, найбільш доцільними з яких на сьогодні є:

а) A-Frame та AR.js – ці API, фактично, унікальні засоби швидкого прототипування, і значна частина програми з їх використанням – це HTML-подібний код, який використовує JavaScript на сервері. A-Frame використовується для створення сцен, об'єктів, анімації та інших 3D-елементів у веб-браузері. AR.js надає можливість відслідковувати маркер і надає можливість сцені, сконструйованій за допомогою A-Frame, відображатися прямо на маркері;

б) Three.js та ARToolKit – своєрідний кістяк, який використовуються багато інших бібліотек мовою JavaScript. Three.js використовує WebGLRenderer, що надає можливість створення якісних 3D-сцени безпосередньо у браузері. На відміну від A-Frame, Three.js використовується в основному для створення веб-програм під управлінням Google Cardboard та вимагає явного використання JavaScript.

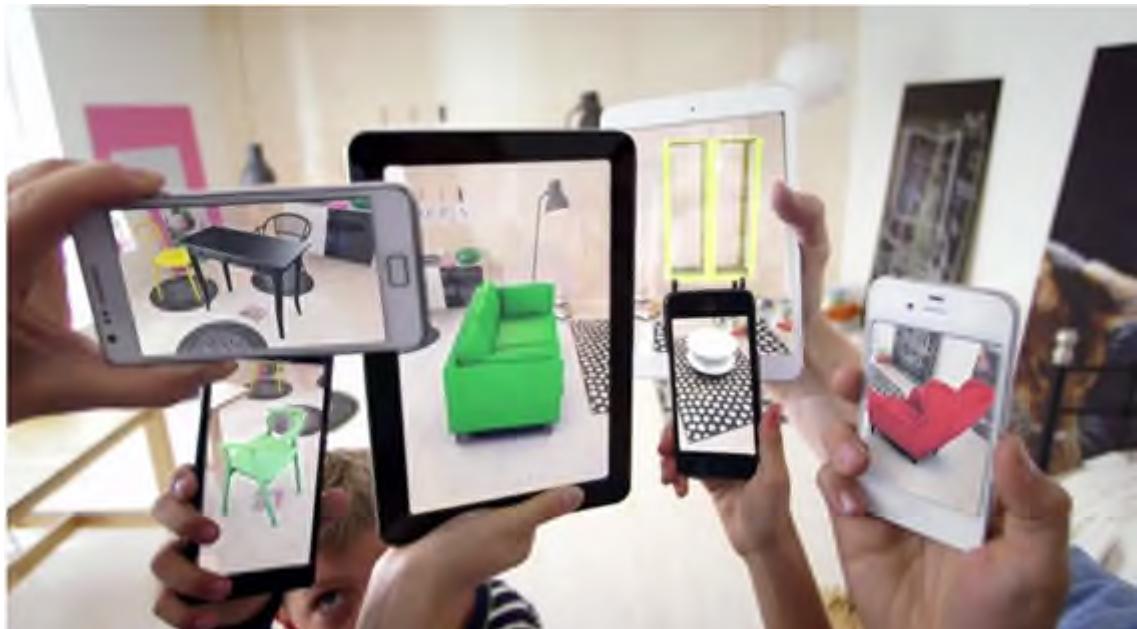
Програмні засоби із доповненою реальністю, розроблені із використанням JavaScript та WebGL, можуть бути розміщені в Інтернет на одному із хмарних сервісів, таких як Heroku.

## 1.2. Як працює доповнена реальність

Для початку роботи необхідно мати лише AR-сумісний браузер, такий як Firefox або Chrome, та текстовий редактор (типу mcedit або Sublime). Базові знання HTML, CSS та JavaScript є обов'язковими, а досвід роботи з веб-API та GitHub стануть у пригоді.

Надалі під AR будемо розуміти здатність пристрою, зокрема мобільного

пристрою або веб-браузера, відстежувати зображення або відображати 3D-об'єкт поверх цього зображення. Головна ідея AR полягає в тому, щоб відобразити комп'ютерну модель у реальному часі та реальному просторі з метою взаємодії між користувачем у реальному просторі та 3D-моделі у віртуальному.



AR може бути як маркерним, так й безмаркерним. У маркерній AR пристрій відстежує 2D-маркер: коли він знаходиться, на ньому фактично відображається 3D-об'єкт. У безмаркерному варіанті пристрій буде шукати плоску поверхню (стіл, підлогу тощо), і розташовуватимемо 3D-об'єкт на ній.

Використовуючи камеру пристрою, AR надає можливості відображення комп'ютерно згенерованих об'єктів в ігрових, маркетингових та інших програмах – наприклад, для розстановки меблів у вітальні або примірки одягу перед їх покупкою. Це дійсно велика можливість для бізнесу – показати, як виглядає продукція, перш ніж будь-який споживач дійсно її купує.

Для AR розробляються спеціальні пристрої, як правило, у вигляді шоломів та гарнітур, що надають можливість занурення користувача у модельне середовище.



### 1.3. Порівняння доповненої та віртуальної реальності

AR доповнює реальний світ 3D-моделями, якими можна керувати за допомогою мобільного пристрою в будь-якому місці. Віртуальна реальність (Virtual Reality – VR) занурює користувача у модельний світ, для чого, як правило, необхідні наголовні дисплеї (Head Mounted Devices – HMD).



Інтерактивність у програмах для AR і VR забезпечується дуже схоже. Так, наприклад, VR фактично використовує контролери, а у деяких випадках

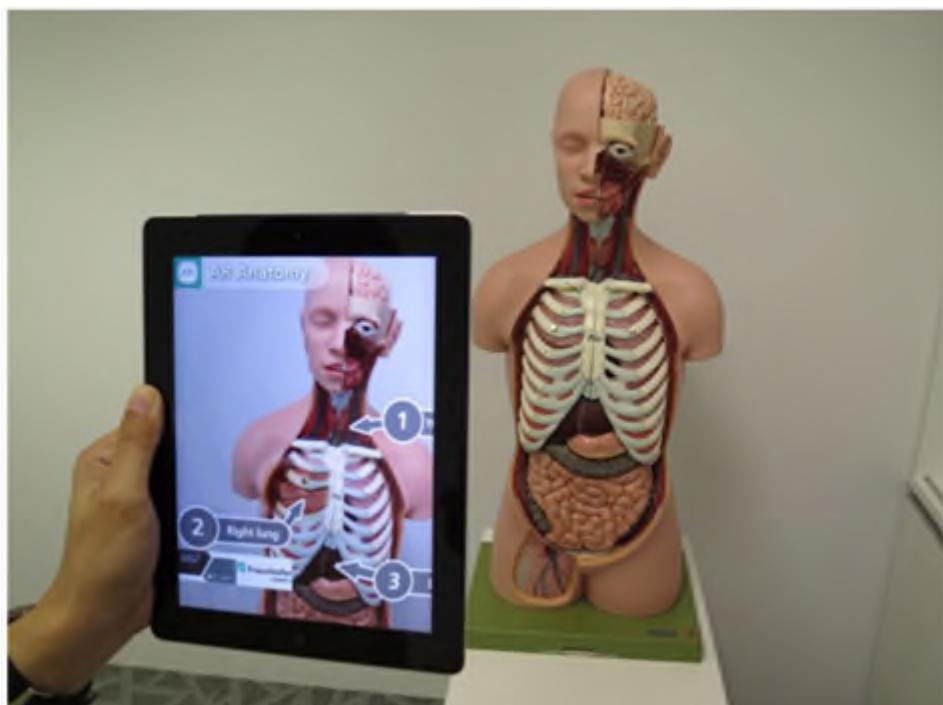
й відстеження рук, що надає змогу користувачеві взаємодіяти з 3D-об'єктами всередині сцени, у якій вони знаходяться.

До головних небезпек використання HMD для роботи у VR відносяться:

- напруження очей;
- запаморочення і головні болі після використання HMD.



На відміну від VR, AR не має таких значних ризиків для здоров'я. Тим не менш, викликає занепокоєння можливість користувачів залишатися зосередженими на тому, що вони роблять, під час використання AR – зокрема, з причин безпеки.



## 1.4. Пристрої доповненої реальності

Найбільш поширений тип пристроїв, готових для AR – смартфони та планшети з операційними системами iOS (версія 11 та вище під управлінням iPhone та iPad) та Android (версія 7 та вище).



Для веб-браузерів AR доступна, якщо у них реалізована підтримка WebRTC та WebGL – насамперед Google Chrome та Mozilla Firefox.

HoloLens є HMD-подібною гарнітурою для AR, що знаходитьться у активній розробці. Цю гарнітуру часто порівнюють з AR-окулярами, такими як CastAR, Meta, Laster SeeThru та K-Glass.



## 2 СТВОРЕННЯ ПРОСТИХ AR-ПРОГРАМ ЗА ДОПОМОГОЮ A-FRAME ТА AR.JS

### 2.1. Налаштування проекту та створення сцени

A-Frame дуже схожий на HTML – усі команди описуються тегами, які подібні до тегів HTML, але, на відміну від останніх, інтерпретуються не у веб-браузері на боці клієнта, а є способом доступу до JavaScript, що виконується на боці сервера. Разом із AR.js він є потужним API для AR, що приховує деталі реалізації мовою JavaScript.

Для початку роботи з A-Frame необхідно перейти на сайт A-Frame за посиланням <https://aframe.io/docs/1.0.0/introduction/installation.html> та завантажити з нього файл збірки JavaScript (JS Build, Production Version – <https://aframe.io/releases/1.0.4/aframe.min.js>).

The screenshot shows the A-Frame website's 'Include the JS Build' section. On the left, there's a sidebar with links like DOCS, BLOG, COMMUNITY, SHOWCASE, GITHUB, SLACK, SUBSCRIBE, ASK A QUESTION, and a VERSION dropdown set to 1.0.0. The main content area has a heading 'Include the JS Build' and a note: 'To include A-Frame in an HTML file, we drop a <script> tag pointing to the CDN build:' followed by a code snippet. Below it, a note says: 'If we want to serve it ourselves, we can download the JS build:' with two buttons: 'Production Version 1.0.3' (Minified) and 'Development Version 1.0.3' (Uncompressed with Source Maps).

Завантажений файл збірки необхідно зберегти у окремому каталогі (наприклад, aframe) всередині робочого каталогу, після чого створити у останньому індексний файл HTML ARindex.html:

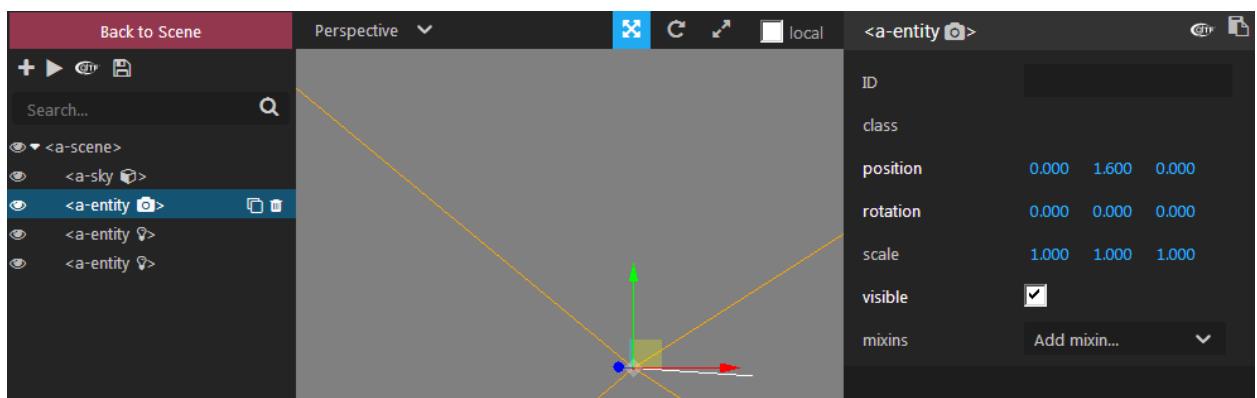
```
code
  ARindex.html
  aframe
    aframe.min.js
  із наступним вмістом
  <!DOCTYPE html>
```

```
<html>
  <script src="aframe/aframe.min.js"></script>
  <a-scene>
    <a-sky color="grey"></a-sky>
  </a-scene>
</html>
```

Файл містить команду `script` із посиланням на завантажений файл – вона завжди використовується для початку роботи з бібліотекою JavaScript. Саме у ньому інтерпретується тег `a-scene`, в якому знаходитьться більша частина коду A-Frame. Тег `a-sky` створюватиме фоновий колір (його також можна застосувати для розміщення 360° зображення на сцені). Після відкриття файлу ARindex.html у веб-браузері отримаємо вікно із сірим фоном.



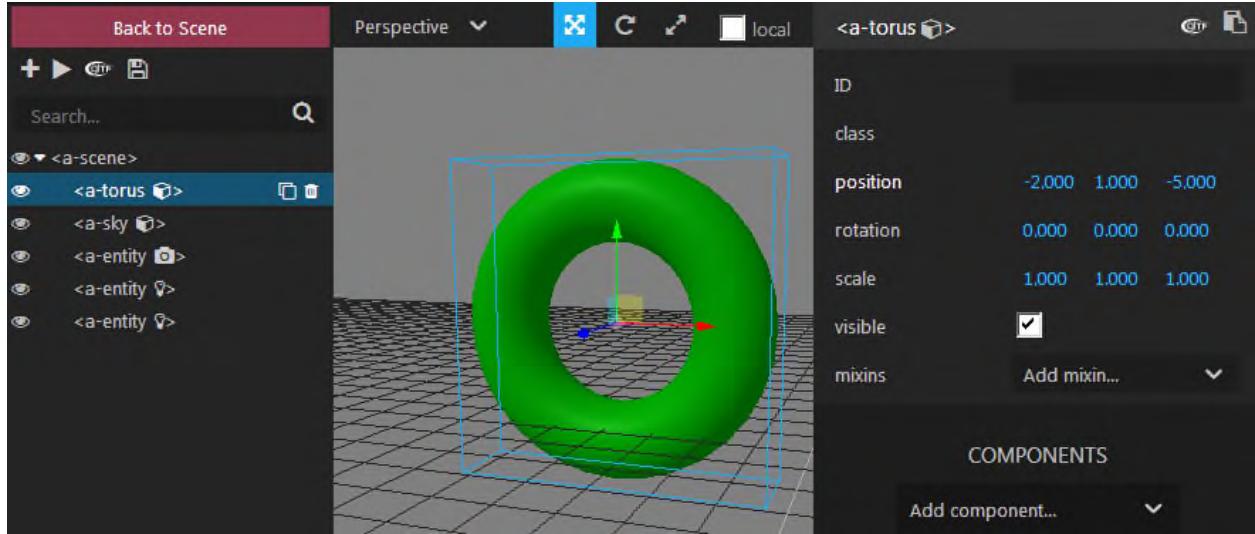
Про те, що A-Frame API дійсно працює, свідчить режим VR (Virtual Reality) у нижньому правому куті. Суттєво більше відомостей можна отримати, увімкнувши режим візуального 3D-інспектора (Ctrl-Alt-I).



Додавши до індексного файлу команду

```
<a-torus position="-2 1 -5" color="green" radius="1.2"></a-torus>
```

отримаємо зображення зеленого тору на сірому тлі.



У новостворених об'єктів є певні атрибути – ознайомитись із ними можна у документації до A-Frame. Поточна версія підтримує наступні примітиви:

a-box – прямокутний паралелепіпед;

a-camera – камера (визначає, що бачить користувач);

a-circle – круг;

a-collada-model – 3D-модель у форматі COLLADA (.dae);

a-cone – конус;

a-cursor – опрацювання подій від миши;

a-curvedimage – панорамне зображення;

a-cylinder – циліндричні поверхні;

a-dodecahedron – дванадцятигранник;

a-gltf-model – 3D-модель у форматі glTF (.gltf);

a-icosahedron – двадцятигранник;

a-image – пласке зображення;

a-light – джерела світла;

a-link – гіперпосилання;

a-obj-model – 3D-модель у форматі Wavefront (.obj/.mtl);

a-octahedron – восьмигранник;  
 a-plane – площа;   
 a-ring – пласке кільце або диск;  
 a-sky – додає фоновий колір або 360° зображення;  
 a-sound – джерело звуку;  
 a-sphere – сфера або багатогранник;  
 a-tetrahedron – трикутна піраміда;  
 a-text – плаский текст;  
 a-torus-knot – кренделеподібна фігура;  
 a-torus – тор;  
 a-triangle – трикутна поверхня;  
 a-video – відео як текстура на площині;  
 a-videosphere – 360° фонове відео.

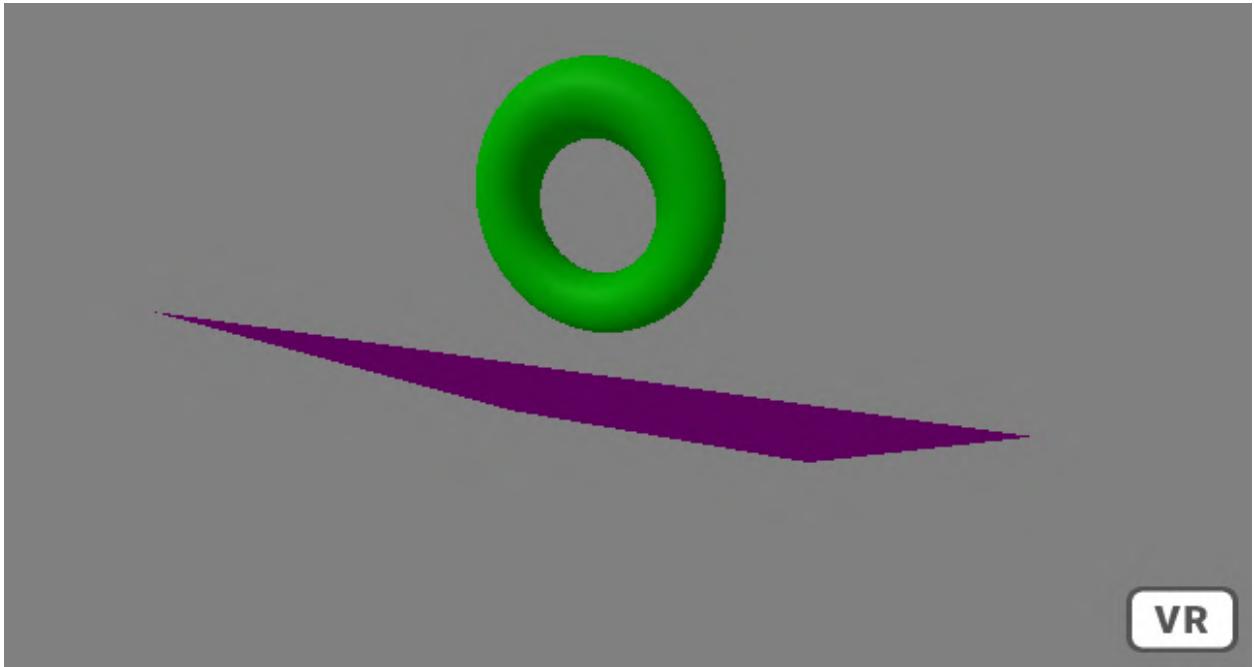
## 2.2. Об'єктні сітки та атрибути сітки

Додамо до попередньої сцени, що містить зелений тор та сірий фон, ще кілька примітивів. Спочатку вставимо між тором та фоном площину:

```
<a-plane width="7" height="7" rotation="50 0 0"
         position="3 -2 -3" color="purple"></a-plane>
```

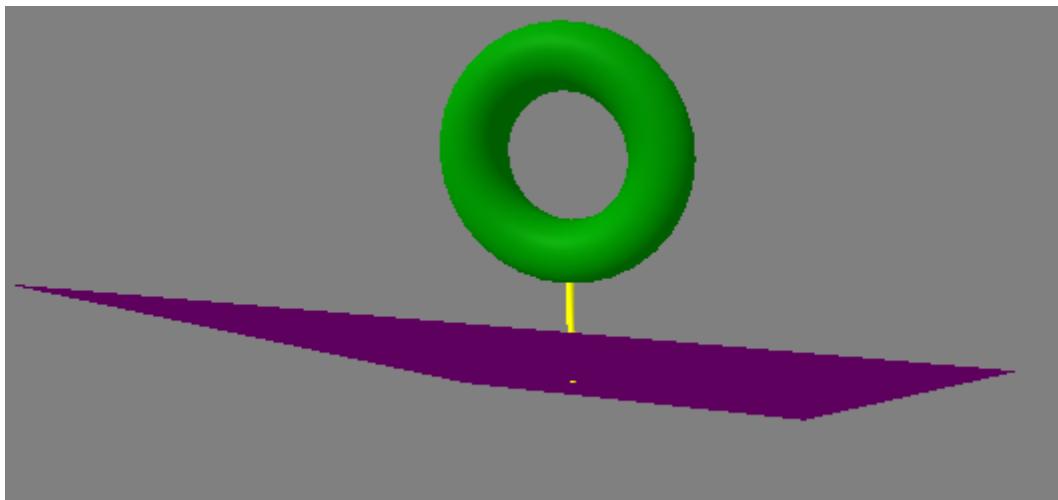
Для перегляду параметрів a-plane слід скористатись документацією за посиланням <https://aframe.io/docs/1.0.0/primitives/a-plane.html> – тут можна знайти різні атрибути площини, які можна застосувати до неї в тегу A-Frame. Параметри width та height відповідають за ширину та висоту прямокутника, rotation вказує на необхідність її повороту на 50° відносно вісі x та на 0 – відносно у та z, position – координати початку площини за відповідними осями, а color – обраний колір.

Площину, налаштовану у такий спосіб, на сцені можна побачити лише в режимі інспектора, тому що вона розміщена дуже далеко на задньому плані. Якщо змінити координати початку площини на "-2 -2 -5", її можна побачити.



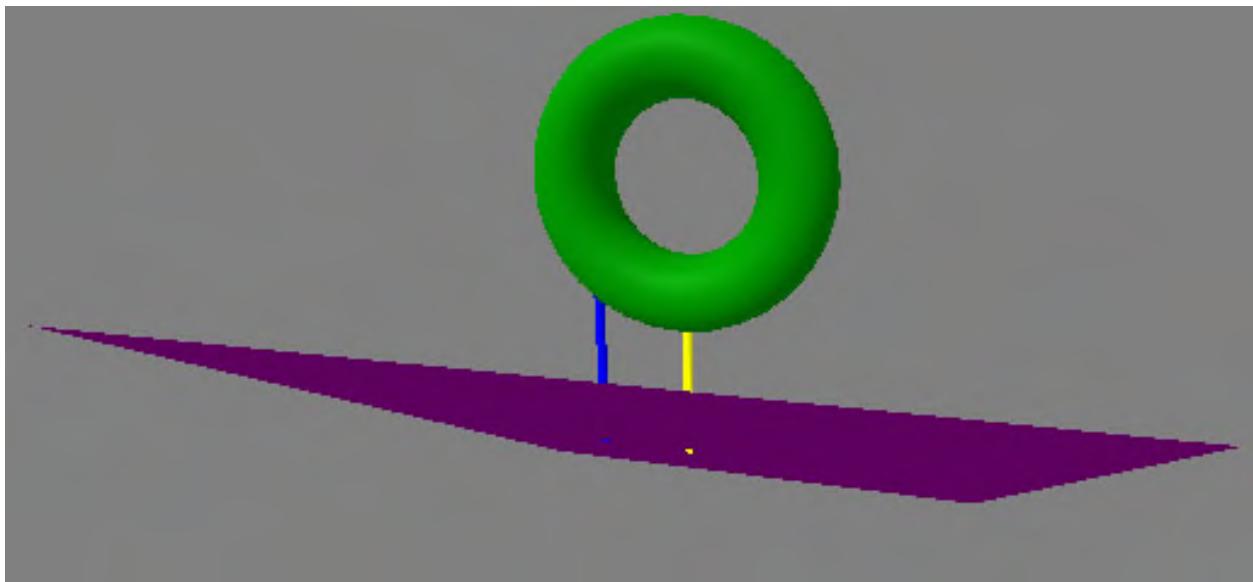
Отже, якщо в певній позиції об'єкт фактично не відображається, це може бути проблемою з позиціонуванням самого об'єкту всередині коду. Тепер, коли у нас є площа, додамо ще кілька об'єктів. Створимо циліндричний об'єкт у формі льодяника жовтого кольору, розташованого під самим тором

```
<a-cylinder color="yellow" height="2" radius="0.05"
    position="-2 -1 -5"></a-cylinder>
```



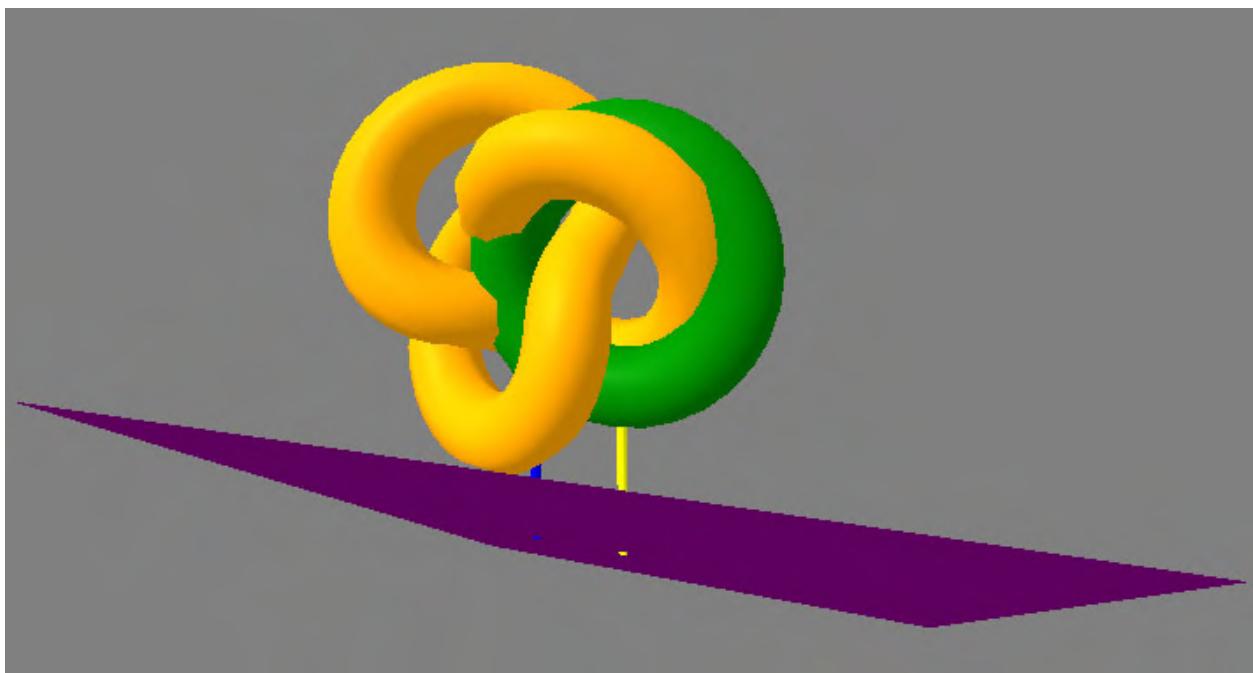
Ще один циліндр:

```
<a-cylinder color="blue" height="2" radius="0.05"
    position="-3 -1 -5"></a-cylinder>
```



I, нарешті, кренделеподібна фігура:

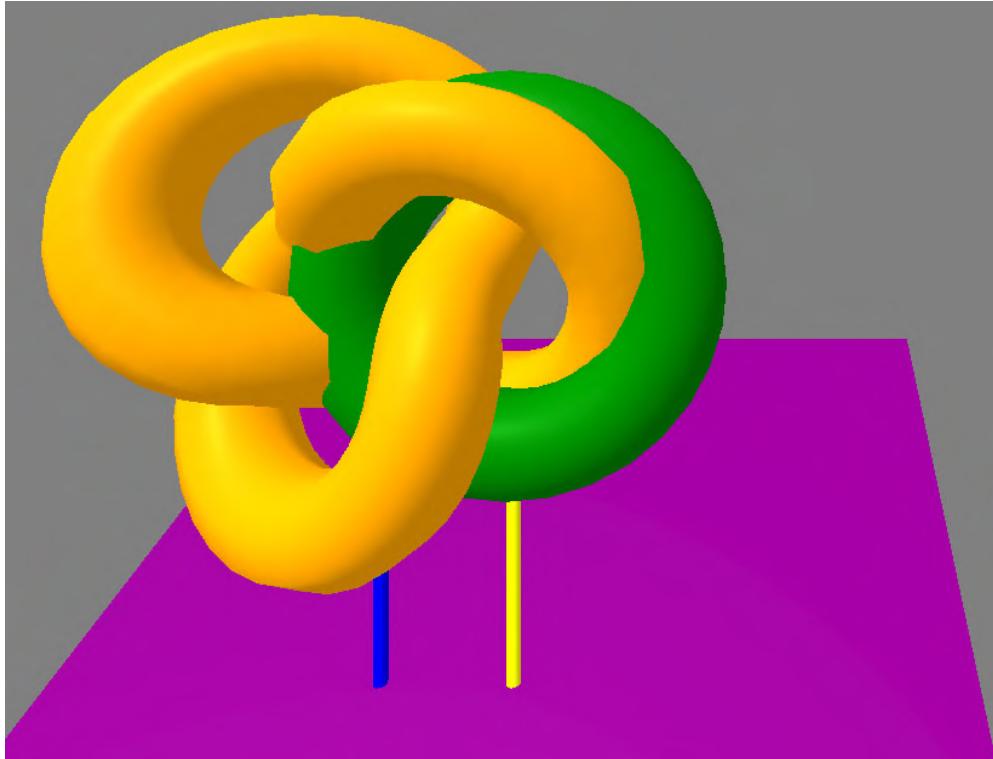
```
<a-torus-knot color="orange" radius="1.2"
position="-3 1 -5"></a-torus-knot>
```



Досить цікава форма – без документації не розібратись.

### **2.3. Додавання тексту та анімації до об'єктів**

Виконаємо невеликі зміни у індексному файлі в атрибуті обертання площини – зміна кута огляду на "-55 0 0" надає їй суттєво кращого вигляду.

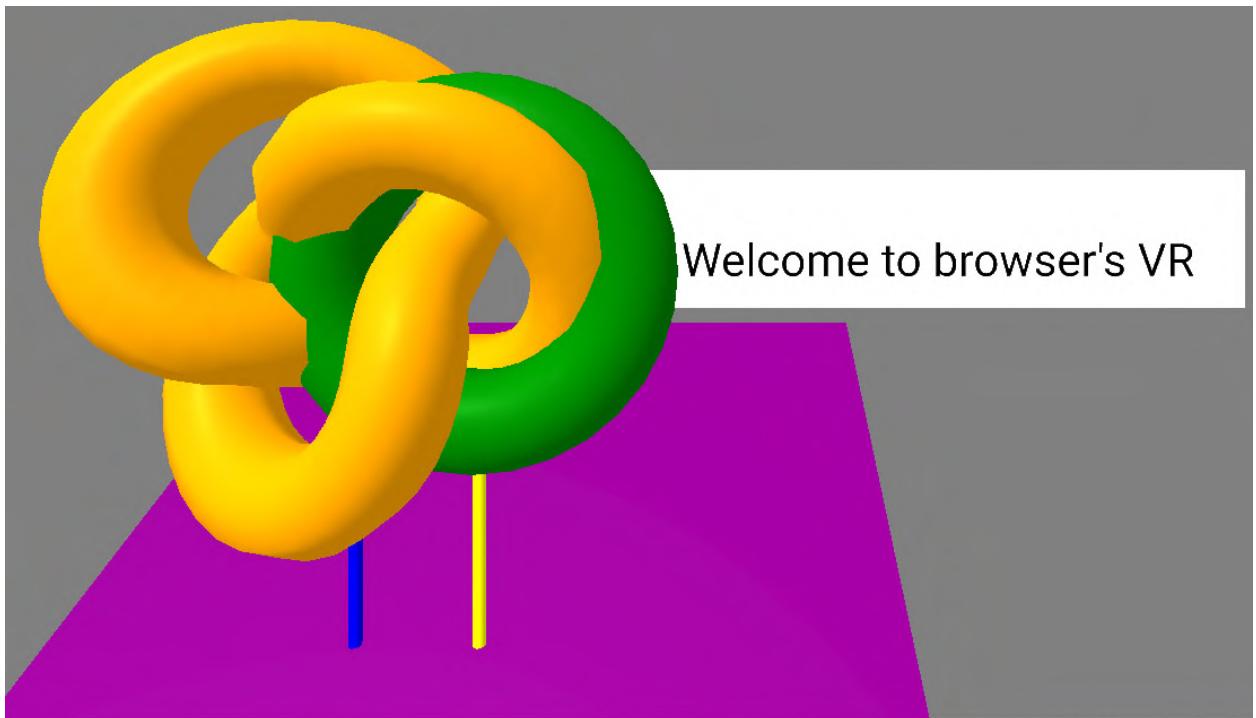


Додамо до сцени площину, перпендикулярну попередній:

```
<a-plane width="9" height="2" position="3 1 -9"></a-plane>
```

За замовчанням її колір буде білим – достатньо зручний для того, щоб перед ним розмістити напис:

```
<a-text value="Welcome to browser's VR" color="black"
       width="10" position="-0.5 1 -6"></a-text>
```



«Лобова» спроба замінити даний напис на кириличний, скоріше за все,

виявиться невдалою без використання відповідного шрифта. Більш ніж 2000 різних шрифтів та способи їх використання можна знайти у репозитарії <https://github.com/etiennepinchon/aframe-fonts>. Будь-який шрифт із репозитарію може бути підключений за посиланням виду [https://raw.githubusercontent.com/etiennepinchon/aframe-fonts/master/fonts/\[FONT\\_NAME\]/\[FONT\\_TYPE\].json](https://raw.githubusercontent.com/etiennepinchon/aframe-fonts/master/fonts/[FONT_NAME]/[FONT_TYPE].json)

Інший варіант – створення власного шрифту за допомогою MSDF font generator:

## MSDF font generator

[SOURCE](#) [ISSUES](#)

### 1. Select font

Default font is Microsoft YaHei, which supports several languages. Optionally, upload another (.ttf) font:

Upload a font: [Browse...](#) arial.ttf

[RESET](#)

### 2. Select character set

```
qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLMNBVCXZMйцукенгшхіфівапроджेचсмитьбю 1234567890--·॥፣۔<>؟"
```

### 3. Create MSDF font

[CREATE MSDF](#)

The generated file will be named [arial-msdf.json](#).

### 4. Preview and download files

[DOWNLOAD](#)

*Preview shows only first five characters of charset.*

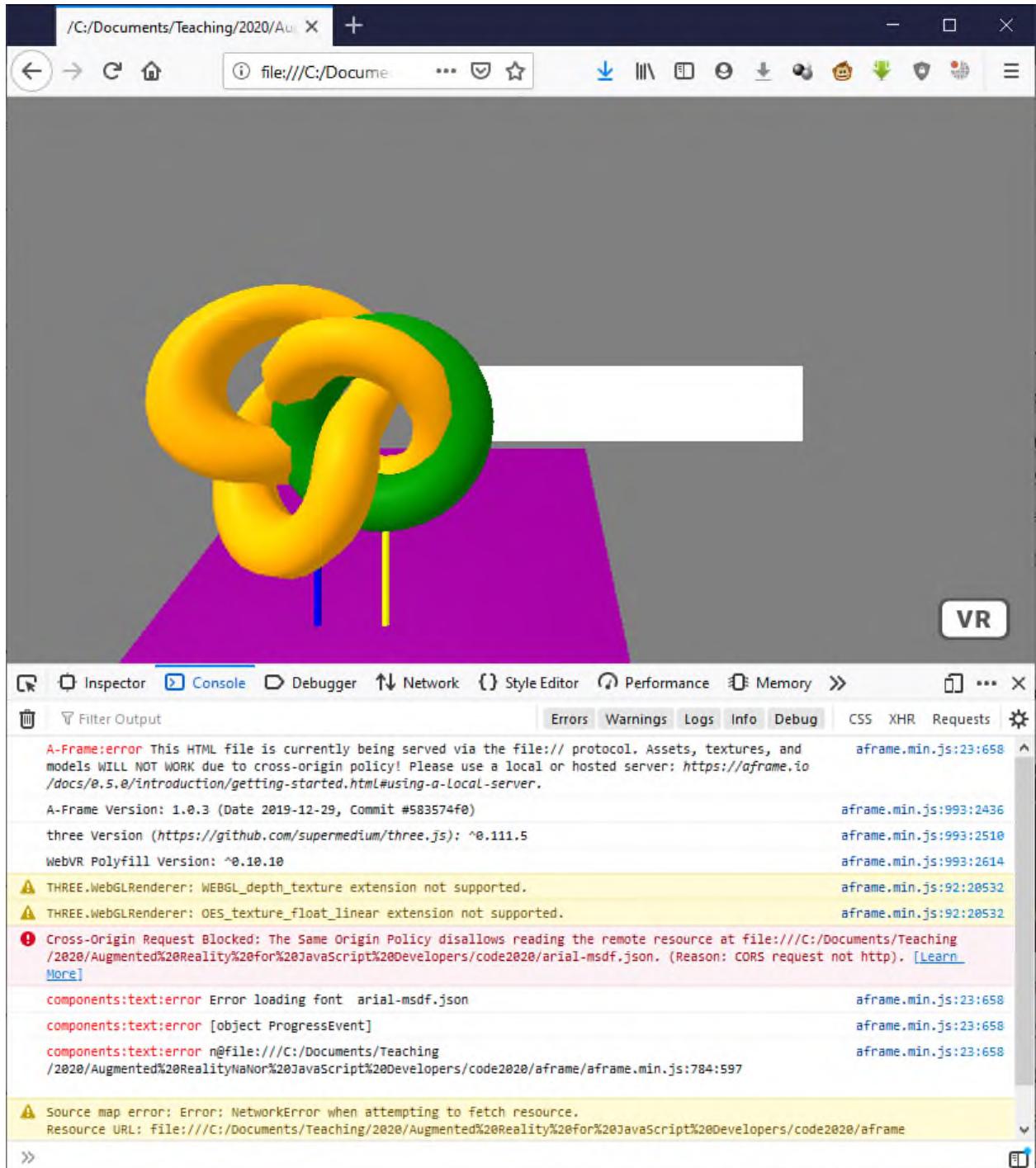
Архів, завантажуваний з <https://msdf-bmfont.donmccurdy.com>, містить шрифт у форматах JSON та PNG (для вихідного шрифту arial.ttf це будуть arial-msdf.json та arial.png відповідно – розташуйте їх там само, де знаходиться й ARindex.html). Чим більше символів обирається, тим більше має бути розмір PNG-файлу (до 1024x1024). Для економії трафіку до них включені лише ті

символи, які були обрані користувачем – всі інші не відображатимуться.

Внесемо зміни до параметру `value` та додамо параметри `font` і `negate`:

```
<a-text value="Бітаємо у браузерній VR!" color="black"
        width="10" position="-0.5 1 -6"
        font="arial-msdf.json" negate="false"></a-text>
```

Якщо останній матиме істинне значення, обраний колір стане фоновим для кожної з літер.



Новий текст не відображається – згідно повідомлень із консолі браузера,

локальне завантаження (за протоколом `file://`) шрифту (так само, як і багатьох інших ресурсів), не дозволено. Для того, щоб виправити цю помилку, необхідно файли шрифту розмістити на сервері з доступом за протоколом HTTP або HTTPS. Де та як це зробити, суттєво залежить від власних можливостей користувача – студенти зазвичай можуть звернутись до викладача для того, щоб отримати доступ до зовнішнього серверу, або налаштувати власний.

Наприклад, для отримання доступу за протоколом FTP до власного розділу на сервері <https://playground2.ccjournals.eu> користувачеві необхідні знати ім'я або адресу FTP-серверу (host, hostname), ім'я (login, username) та пароль (password).

**net2ftp a web based FTP client**

- [HOME](#)
- [SCREENSHOTS](#)
- [FEATURES »](#)
- [DOWNLOAD](#)
- [HELP »](#)
- [ABUSE](#)
- [PRIVACY »](#)

Home

## Login

Connect to your FTP server and start editing your website now.

Basic FTP login

FTP server: ftp.ccjournals.eu

Port: 21

Username: student01@playground2.ccjournals.eu

Password: \*\*\*\*\*

Privacy notices

Please enter your email address as identifier to give you the right of access and erasure:

 Enter your email address

I agree to the [Disclaimer](#), [Privacy Policy](#) and [Cookie Policy](#)

**Login**

Після реєстрації користувач побачить поточний вміст його розділу на сервері:

ftp.ccjournals.eu

Heart

/

Directory Tree: root /

Files which are too big can't be downloaded, uploaded, copied, moved, searched, zipped, unzipped, viewed or edited; they can only be renamed, chmodmed or deleted.

New dir New file Upload Java Upload Install Advanced  
Transform selected entries: Copy Move Delete Rename Chmod Download Zip Unzip Size Search

All	Name	Type	Size	Owner	Group	Perms	Mod Time	
	Up...							
<input type="checkbox"/>	AR.js-master	Directory	4096	ccjourna	ccjourna	rwxr-xr-x	Feb 15 13:31	
<input type="checkbox"/>	aframe	Directory	27	ccjourna	ccjourna	rwxr-xr-x	Feb 15 13:30	
<input type="checkbox"/>	three.js-dev	Directory	316	ccjourna	ccjourna	rwxr-xr-x	Feb 15 13:34	
<input type="checkbox"/>	_ftpquota	FTPQUOTA File	15	ccjourna	ccjourna	rw-----	Feb 15 14:18	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Open</a>
<input type="checkbox"/>	AR.js-master.zip	Zip archive	136626749	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">Open</a>
<input type="checkbox"/>	ARindex.html	HTML file	179	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Open</a>
<input type="checkbox"/>	index.html	HTML file	831	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Open</a>
<input type="checkbox"/>	photo.jpg	JPEG file	39204	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Open</a>
<input type="checkbox"/>	test.html	HTML file	403	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Open</a>
<input type="checkbox"/>	three.js-dev.zip	Zip archive	266302404	ccjourna	ccjourna	rw-r--r--	Feb 15 13:30	<a href="#">Open</a>

У файлі index.html доцільно розмістити посилання на інші файли, створені користувачем. Редагувати їх можна як безпосередньо на сервері, так й локально із подальшим завантаженням. Для того, щоб поточний приклад був працездатним, необхідно завантажити як сам ARindex.html, так й конвертовані файли із шрифтами arial-msdf.json та arial.png:

Upload files and archives

Upload to directory: /

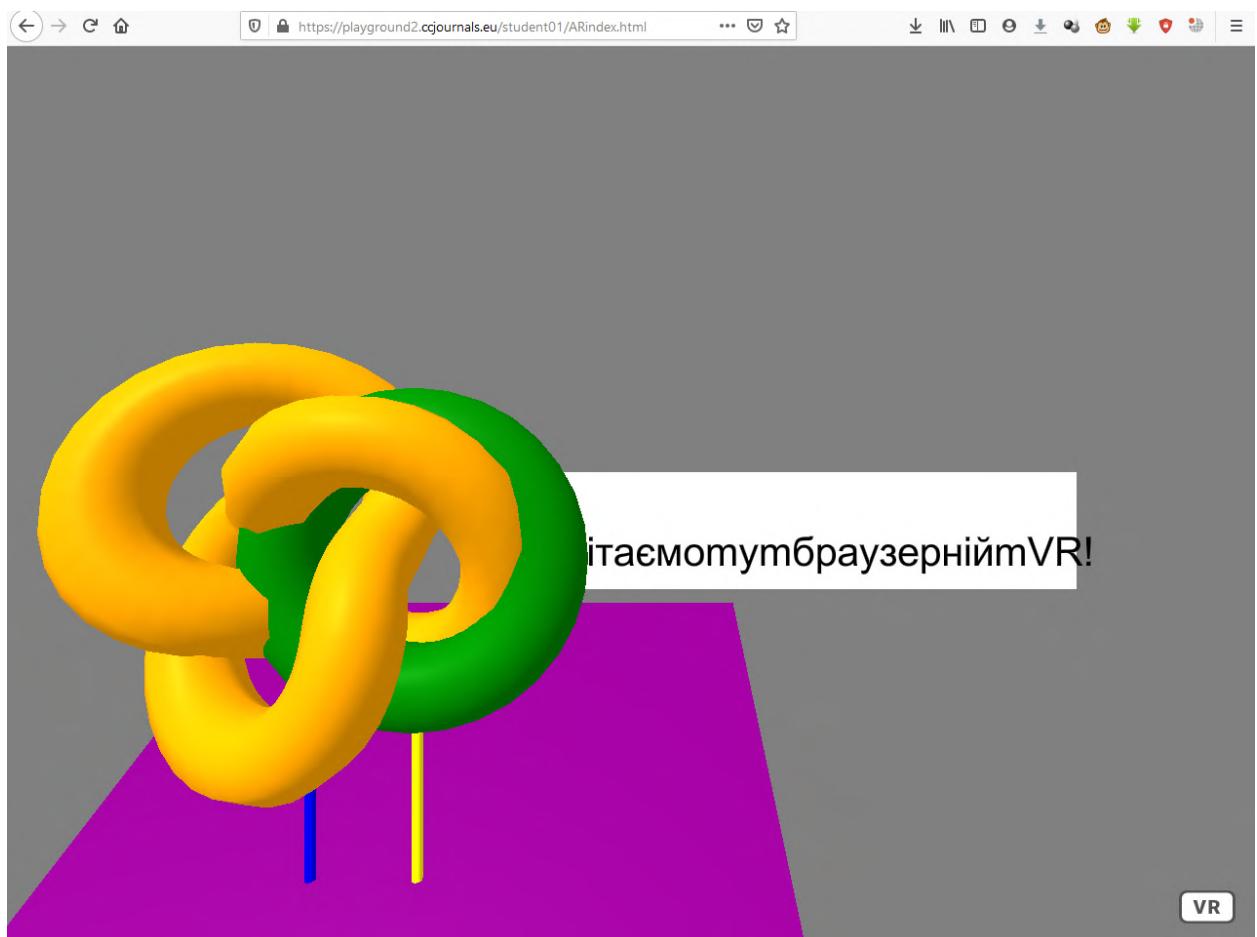
Files  
Files entered here will be transferred to the FTP server.

Archives (zip, tar, tgz, gz)  
Archives entered here will be decompressed, and the files inside will be transferred to the FTP server.

<input type="button" value="Browse..."/> ARindex.html	<input type="button" value="Browse..."/> No file selected.
<input type="button" value="Browse..."/> arial-msdf.json	<input type="button" value="Add another"/>
<input type="button" value="Browse..."/> arial.png	
<input type="button" value="Browse..."/> No file selected.	
<input type="button" value="Add other"/>	

Для масового завантаження файлів доцільно скористатись можливістю завантаження архіву з ними.

Перевірка виконання ARindex.html на сервері показує, що результат вже суттєво кращий:



Для завершення локалізації надпису необхідно масштабувати його до розміру площини, вказавши параметр `scale`, та додати до набору символів, для яких генерується шрифт, пропуск. Застосування символів, що не входять до набору ASCII, потребує додаткового вказання одного з найпоширеніших кодувань тексту – UTF-8 – після тегу `<html>`:

```
<meta charset="utf-8">
```

Для того, щоб анімувати примітив, необхідно додати до нього параметр `animation`.

```
<a-torus position="-2 1 -5" color="green" radius="1.2"
    animation="property: components.material.material.color;
    type: color; from: green; to: red; loop: true; dur: 10000">
</a-torus>
```

У даному прикладі основним в параметрі `animation` є `property` – та складова об'єкту, яка буде змінюватись

(components.material.material.color – колір). dur визначає тривалість анімації в мілісекундах (10000 мс = 10 с), from задає початкове значення атрибуту, to – кінцеве, а loop визначає кількість повторень циклу зміни (true відповідає нескінченному циклу).

Наступний приклад демонструє зміну атрибуту rotation для того, щоб змусити тороподібний об'єкт обертатися навколо центру:

```
<a-torus-knot color="orange" radius="1.2" position="-3 1 -5"
    animation="property: rotation; to: 0 0 360; loop: true; dur: 10000">
</a-torus-knot>
```

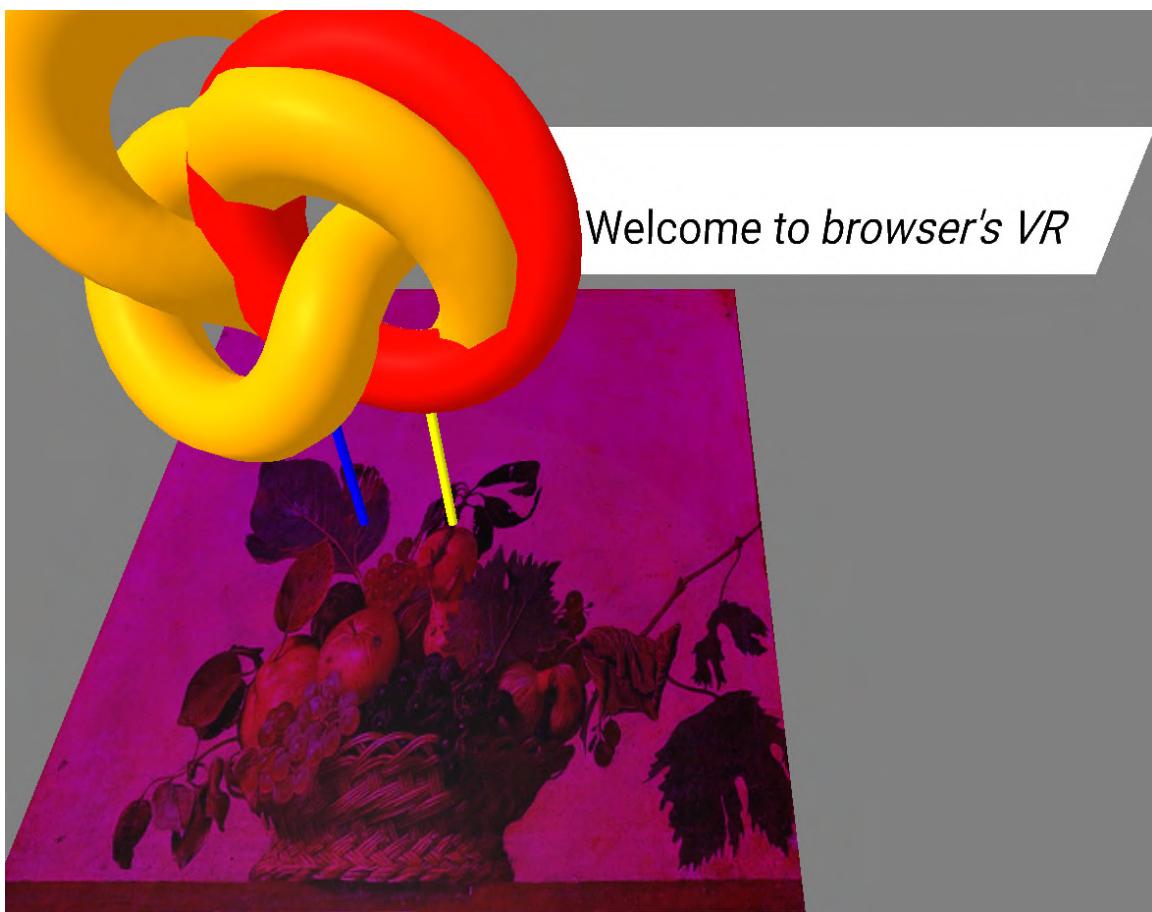
Більш детально про параметр animation можна прочитати у довідці з A-Frame Core API за посиланням <https://aframe.io/docs/1.0.0/components/animation.html>

## 2.4. Додавання текстур до об'єктів

Головна сторінка <http://aframe.io> містить посилання на вихідні коди бібліотеки. На рівень вище серед 18 репозиторіїв A-Frame можна знайти sample-assets – приклади файлів зображень, моделей та аудіозаписів для використання у A-Frame. Для того, щоб скористатися ними, необхідно отримати пряме посилання на об'єкт репозитарію, та додати його у якості атрибуту src до об'єкту, для якого обране зображення виступатиме текстурою. Змінимо площину під торами:

```
<a-plane
src="https://raw.githubusercontent.com/aframevr/sample-
assets/master/assets/images/illustration/758px-
Canestra_di_frutta_(Caravaggio).jpg" width="7" height="7"
rotation="-55 0 0" position="-2 -2 -5" color="purple"></a-plane>
```

Посилання на файл текстури може бути взяте із будь-якого місця, але не завжди сайти дозволяють використовувати прямі посилання на файли сайту поза його межами. «Політика того ж походження» (same origin policy) є важливим механізмом безпеки у сучасних веб-браузерах, що стосується як виконуваних у браузері файлах, так й використовуваних.



## 2.5. Додавання AR.js до програми

Перш за все необхідно перейти до сховища AR.js у GitHub за посиланням <https://github.com/jeromeetienne/AR.js> і завантажити ZIP-файл, натиснувши «Clone or download». Станом на початок 2020 року розмір репозитарію AR.js – більше 170 Мб, тому завантаження архіву може тривати певний час.

**jeromeetienne / AR.js**

Watch 579 ⚡ Star 14.7k Fork 2k

Code Issues 85 Pull requests 2 Actions Projects 1 Wiki Security Insights

Efficient Augmented Reality for the Web - 60fps on mobile!

a-frame three-js webar

1,589 commits 16 branches 0 packages 37 releases 45 contributors MIT

Branch: master New pull request Find file Clone or download

Author	Commit Message	Date
nicolocarpignoli	Last commit before release	Latest commit 32532d7 12 days ago
.github/ISSUE_TEMPLATE	Update issue templates	13 months ago
aframe	Last commit before release	12 days ago
data	clean a lot of old stuff	2 months ago
test	changed THREE.AxisHelper to THREE.AxesHelper in examples	10 months ago
three.js	Last commit before release	12 days ago

Завантажений архів доцільно розпакувати у каталог, де зберігаються індексний файл та каталог aframe:

```

    ARindex.html

    aframe
        aframe.min.js

    AR.js-master
        .gitignore
        .travis.yml
        AR.js-1920-1080-HD.png
        CHANGELOG.md
        CODE_OF_CONDUCT.md
        CONTRIBUTING.md
        HOW_TO_RELEASE.md
        ISSUE_TEMPLATE.md
        LICENSE.txt
        Makefile
        package-lock.json
        package.json
        PULL_REQUEST_TEMPLATE.md
        README.md
        TODO.md

        .github
            ISSUE_TEMPLATE
                bug_report.md

    ...

```

Структура каталогів репозитарію досить розгалужена, але прозора. Так, у AR.js-master\data\images\ можна знайти файл HIRO.jpg – дане зображення буде маркером, необхідним для того, щоб при наведенні на нього веб-камери відображалась сцена.



Для початку використання AR.js необхідно внести кілька змін до вихідного індексного файлу ARindex.html. По-перше, після тегу script для підключення стандартної версії бібліотеки A-Frame додається ще один тег для роботи із AR.js та захоплення даних із веб-камери:

```
<script src="AR.js-master/aframe/build/aframe-ar.min.js"></script>
```

Для того, щоб побудовані об'єкти краще відображались на маркері, змінимо деякі їх атрибути:

- позицію тора встановимо у  $(0; 0,5; 0)$ , а радіус – у  $0,5$ ;
- ширину та висоту площини зменшимо до  $3,5$ , а позицію встановимо у  $(0; -1; 0)$ ;
- позицію першого циліндра змінімо на  $(0; 0; 0)$ , а другого – на  $(1; 0; 0)$ ;
- для кренделеподібної фігури радіус встановимо у  $0,5$ , а позицію – у  $(1; 0,5; 0)$ ;
- до площини, на яку накладатиметься текст, застосуємо текстуру за посиланням [https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/uvgrid/UV\\_Grid\\_Sm.jpg](https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/uvgrid/UV_Grid_Sm.jpg) та встановимо її ширину у  $2,5$ , висоту – в  $1,5$ , а позицію – у  $(0; 1; -1)$ ;
- ширину тексту встановимо у  $3$ , а позицію – у  $(-1; 0,5; -1)$ .

Створена сцена була побудована для використання у веб-VR. Для використання об'єктів на ній у доповненій реальності спочатку позбавимось сірого фону, доданого тегом `a-sky`, шляхом його видалення. Далі необхідно вбудувати у сцену AR.js шляхом додавання атрибутів `embedded` (застосовувати вбудований розпізнавач маркерів) та `arjs = 'trackingMethod: best'` (використовувати найкращий метод відстеження):

```
<a-scene embedded arjs = 'trackingMethod: best;'>
```

Крім того, доведеться створити тег `a-anchor` для того, щоб виконати прив'язку до доповненої реальності:

```
<a-anchor hit-testing-enabled='true'>
```

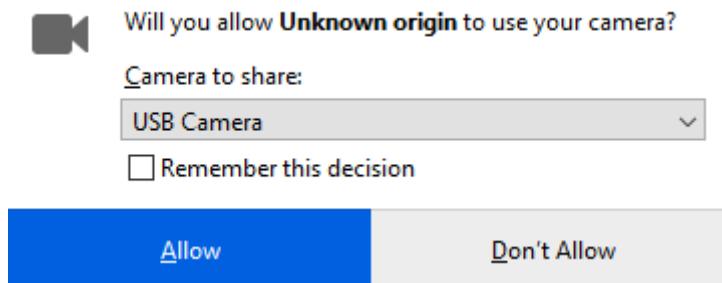
Всі об'єкти, розміщені на сцені, слід розмістити між тегами `<a-anchor>` та `</a-anchor>`. Останній крок – додавання статичної камери:

```
<a-camera-static/>
```

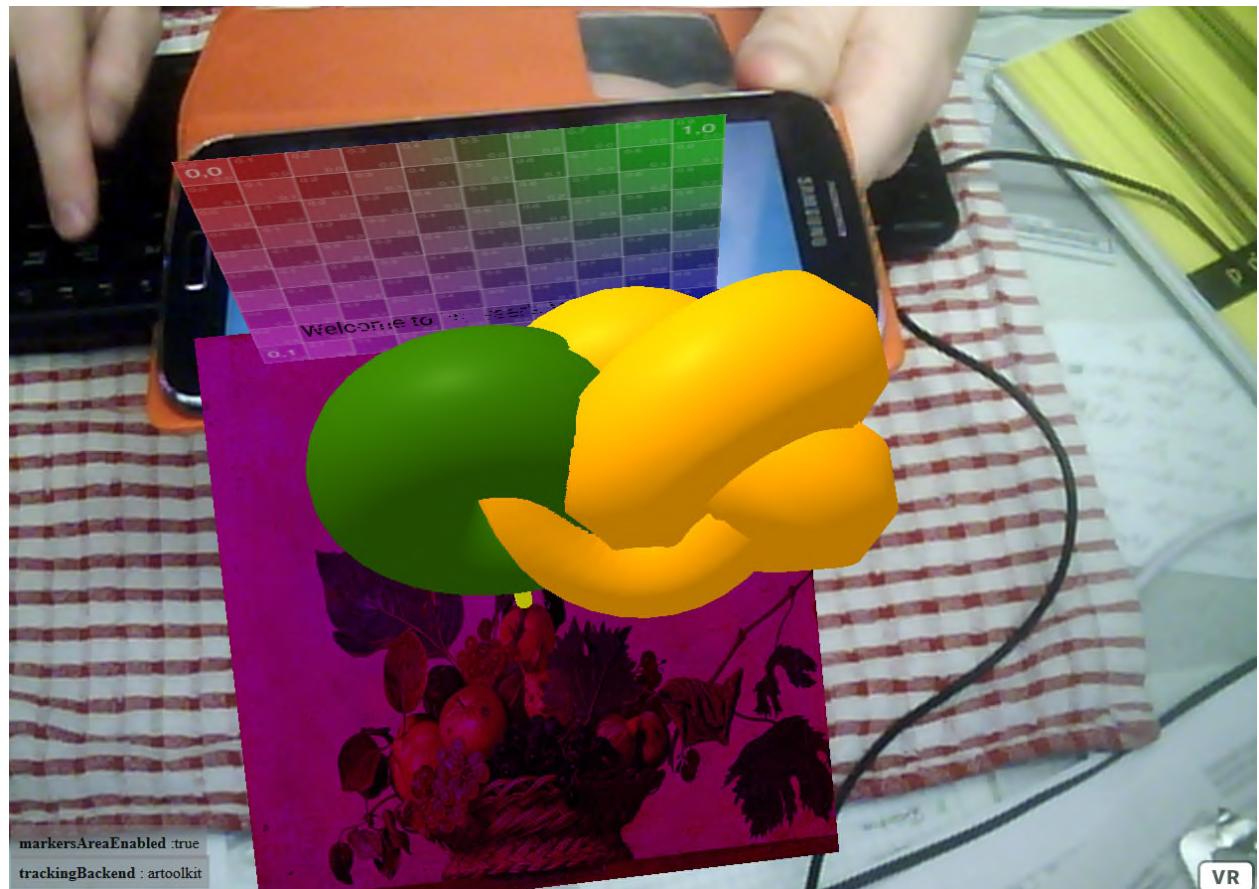
На відміну від попередніх, це тег не є парним, що, за стандартом XHTML, потребує додавання слешу наприкінці тега.

```
<!DOCTYPE html>
<html>
  <meta charset="utf-8">
  <script src="aframe/aframe.min.js"></script>
  <script src="AR.js-master/aframe/build/aframe-ar.min.js"></script>
  <a-scene embedded arjs='trackingMethod: best;'>
    <a-anchor hit-testing-enabled='true'>
      <a-torus position="0 0.5 0" color="green" radius="0.5"
        animation="property: components.material.material.color; type: color; from: green; to: red; loop: true; dur: 10000"></a-torus>
      <a-plane width="3.5" height="3.5" rotation="-55 0 0" position="0 -1 0" color="purple" src=
      "https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/illustration/758px-Caravaggio_di_frutta_(Caravaggio).jpg"
      ></a-plane>
      <a-cylinder color="yellow" height="2" radius="0.05" position="0 0 0"></a-cylinder>
      <a-cylinder color="blue" height="2" radius="0.05" position="1 0 0"></a-cylinder>
      <a-torus-knot color="orange" radius="0.5" position="1 0.5 0" animation="property: rotation; to: 0 0 360; loop: true; dur: 10000">
      </a-torus-knot>
      <a-plane width="2.5" height="1.5" position="0 1 -1"
      src="https://raw.githubusercontent.com/aframevr/sample-assets/master/assets/images/uvgrid/UV_Grid_Sm.jpg"></a-plane>
    <!--
      <a-text value="Вітаємо у браузерній VR!" color="black" width="10" position="-0.5 1 -6"
        font="arial-msdf.json" negate="false"></a-text> --
      <a-text value="Welcome to browser's VR" color="black" width="3" position="-1 0.5 -1"></a-text>
    </a-anchor>
    <a-camera-static/>
  </a-scene>
</html>
```

Після відкриття індексного файлу у веб-браузері, за наявності вебкамери, необхідно надати дозвіл на доступ до неї.



Об'єкти на сцені з'являться, коли камера буде спрямована на маркер.



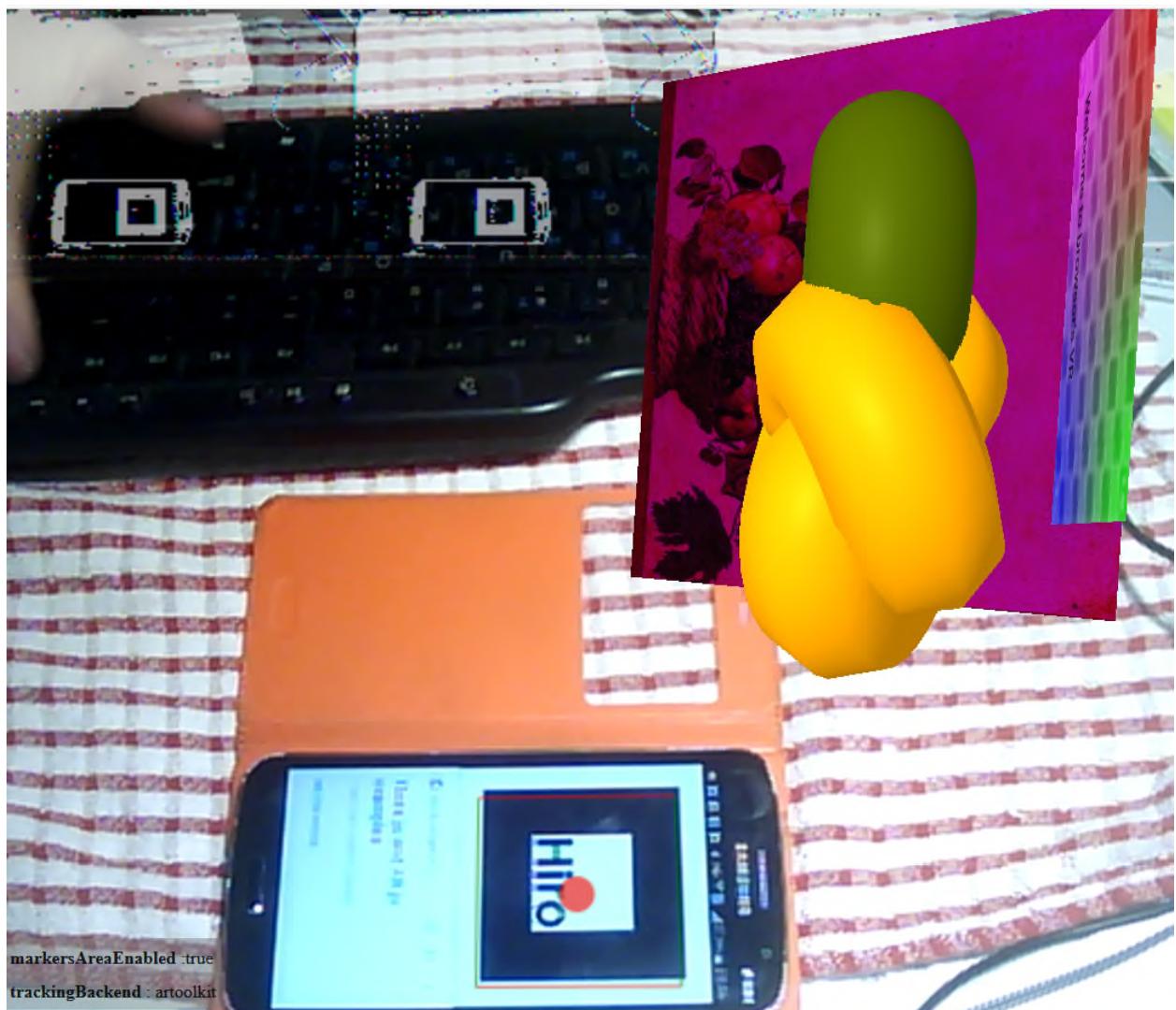
Файл маркеру за замовчанням можна відкрити на будь-якому мобільному пристрой або роздрукувати.

До параметру `args` можуть бути передані наступні значення:

`trackingMethod` – спосіб відстежування маркерів (за замовчанням `'best'`);

`debugUIEnabled` – показувати додаткові відомості для налагодження (за замовчанням `true`);

`debug` – вмикає налагоджувальний режим (за замовчанням `false`);



`detectionMode` – тип маркеру (можливі `'color'`, `'color_and_matrix'`, `'mono'`, `'mono_and_matrix'`);

`matrixCodeType` – тип матричного коду для `detectionMode` `'color_and_matrix'` та `'mono_and_matrix'` (можливі `3x3`, `3x3_HAMMING63`, `3x3_PARITY65`, `4x4`, `4x4_BCH_13_9_3`, `4x4_BCH_13_5_5`);

`patternRatio` – співвідношення сторін для користувальників маркерів (за замовчанням не використовується: -1);

`cameraParametersUrl` – посилання на параметри камери (наприклад, "AR.js-master\data\data\camera\_para.dat");

`maxDetectionRate` – максимальна частота, з якою бібліотека намагається знайти маркер на зображені з камери (за замовчанням не використовується: -1);

`sourceType` – джерело зображення (можливі 'webcam', 'image', 'video');

`sourceUrl` – посилання на джерело (якщо `sourceType` – 'image' або 'video');

`sourceWidth` та `sourceHeight` задають роздільну здатність вихідного зображення (наприклад, 640x480), `displayWidth` та `displayHeight` – відповідно відображуваного зображення;

`deviceId` – опціональний ідентифікатор камери;

Більшість із цих значень використовує найстаріша бібліотека для доповненої реальності – ARToolKit, явне використання якої може бути задано параметром `artoolkit` примітиву `a-scene`. Детальні керівництва та приклади створення веб-AR за допомогою AR.js можна знайти у репозитарії його автора – Жерома Етьєнна.

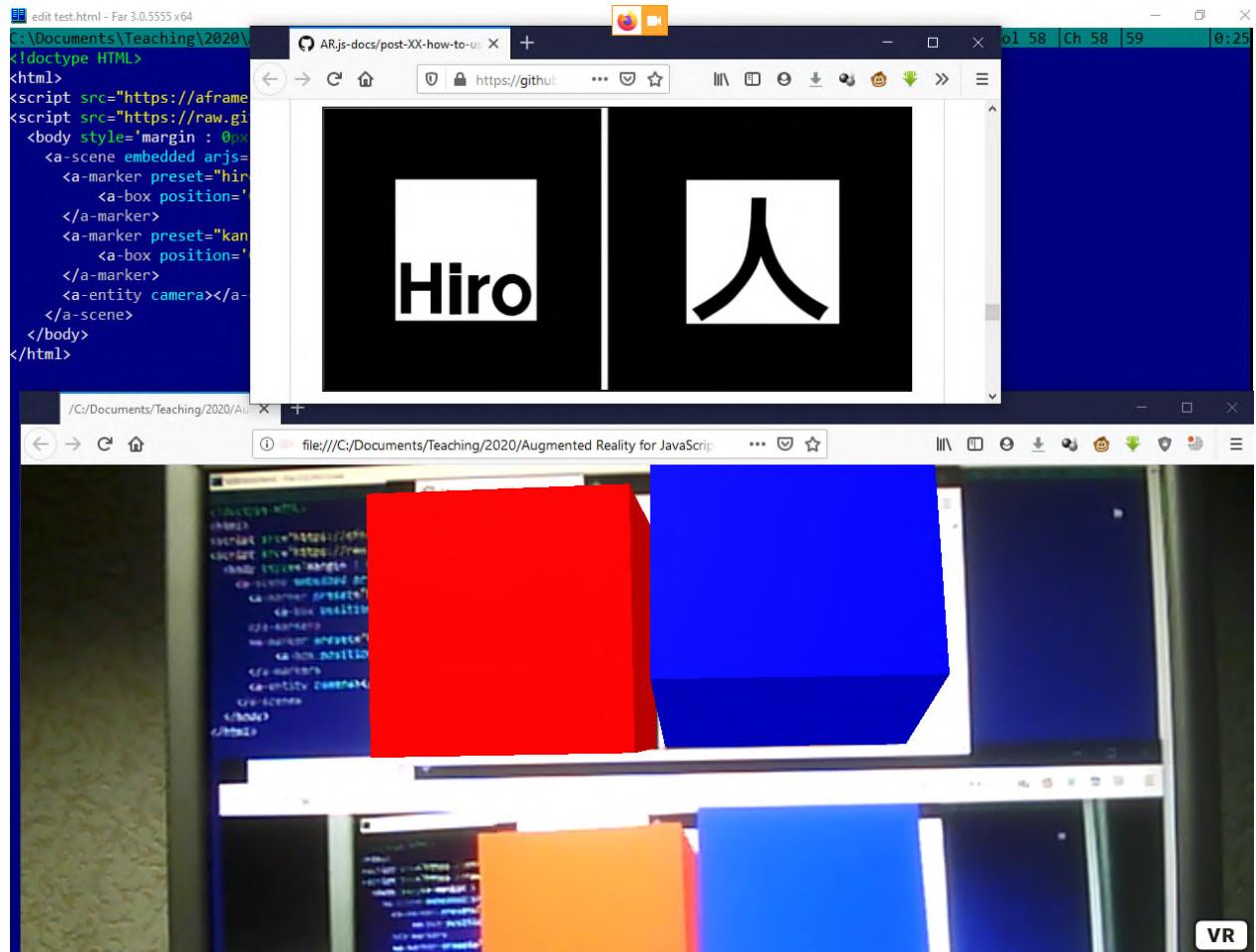
## 2.6. Базовий JavaScript та об'єктна модель A-Frame

Використовуючи тег `script` із атрибутом `src`, бібліотеки (і взагалі будь-який текст мовою JavaScript) можна включати до індексного файлу як з локального розташування, так і з віддаленого. Наприклад, замість завантаження бібліотек A-Frame та AR.js можна було б послатися на їх розташування в мережі Інтернет. Якщо тег `a-anchor` виконує загальну прив'язку групи об'єктів, що він їх охоплює, до маркеру, то у даному прикладі явно вказано, який маркер буде використовуватись, за допомогою тегу `a-marker`: у такий спосіб різним маркерам можна співставити різні групи

об'єктів:

```
<!doctype HTML>
<html>
<script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
<script src="https://raw.githack.com/jeromeetienne/AR.js/2.1.8/aframe/build/aframe-ar.js"></script>
<body style='margin : 0px; overflow: hidden;'>
  <a-scene embedded arjs="debugUIEnabled:false">
    <a-marker preset="hiro">
      <a-box position='0 0.5 0' material='color: red;'></a-box>
    </a-marker>
    <a-marker preset="kanji">
      <a-box position='0 0.5 0' material='color: blue;'></a-box>
    </a-marker>
    <a-entity camera></a-entity>
  </a-scene>
</body>
</html>
```

У даному прикладі використовуються два стандартні маркери – "hiro" та "kanji".



Крім того, одинарний тег `<a-camera-static/>` замінено на об'єкт `camera` за допомогою пари тегів `<a-entity camera></a-entity>`. Зауважимо, що, хоча у цьому прикладі різниця несуттєва, дана заміна не є однозначною. Головна її мета – показати, що JavaScript є об'єктно-

орієнтованою мовою. Будь-який приклад, з якого розпочинається використання цієї мови, яскраво це ілюструє:

```
<!DOCTYPE html>
<html> <body>
    <script type="text/javascript">
        document.write("<font size=7>Hello World!</font>") ;
    </script>
</body>
</html>
```

Тег `script` доповнений атрибутом `type`, у якому конкретизовано мову. Програма складається з одного єдиного рядка, в якому з об'єкту `document` викликається метод `write`, параметром якого є рядок, що з'являється у вікні браузера.

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. *Коментарі* створюються аналогічно до C++:

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

### 3. Прості типи даних:

– рядковий (`string`) – визначається подвійними або одинарними лапками і використовується для символних 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`;

– символний (`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 остатчу 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 первого `case`, виконаються всі наступні та `default`. `default` виконається 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 вище, 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-uEpvTa1bHDNlxUL2klHUMHJu>).

A-Frame використовує ECS (Entity – Component – System) – шаблон проектування комп'ютерних ігор, основними поняттями якого є Entity (сущність), Component (компонент) та System (система). Сущність – це контейнер для компонентів. Сущності є основою всіх об'єктів на сцені, але без компонентів сущності нічого не роблять і не надають. Компонент – це невеликий об'єкт, який реалізує певну структуру даних та відповідає за окрему частину логіки роботи програми. Кожен тип компонента можна прикріпити до сущності, щоб надати їй певної властивості. Системи управлюють набором сущностей, об'єднаних деякими компонентами. Вони не є обов'язковими.

У A-Frame цей шаблон проектування реалізований за допомогою атрибутів. Як сущностей використовуються будь-які примітиви A-Frame – a-scene, a-box, a-sphere та ін. Але особливе місце займає a-entity, ім'я говорить саме за себе. Всі інші примітиви є по суті обгортками для компонентів і зроблені для зручності, тому що будь-який елемент можна створити і за допомогою a-entity. Наприклад, примітив a-box можна реалізувати у такий спосіб:

```

<a-entity
  geometry="primitive: box; width: 1; height: 1; depth: 1">
</a-entity>

```

geometry у даному випадку є компонентом, який був доданий до сущності <a-entity>. Сам по собі <a-entity> не має будь-якої логіки (в глобальному сенсі), а компонент geometry по суті перетворює його на куб або

що-небудь інше. Іншим, не менш важливим, ніж `geometry`, компонентом є `material`. Він додає до геометрії матеріал. Матеріал відповідає за те, чи буде наш куб блищати як метал, чи буде мати будь-які текстури та ін.

Будь-компонент у A-Frame повинен бути зареєстрований глобально через спеціальну конструкцію:

```
AFRAME.registerComponent('hello-world', {
  init: function () {
    console.log('Hello, World!');
  }
});
```

Створений компонент можна додати на сцену, як і будь-який інший елемент:

```
<a-entity hello-world></a-entity>
```

Створення компоненту приводить до виклику методу `init`, що виводить повідомлення до консолі веб-браузера. У компоненті можуть бути визначені також методи:

`update` – викликається при ініціалізації разом з `init` та при оновленні будь-якого властивості компонента;

`remove` – викликається після видалення компонента або сутності, що його містить;

`tick` – викликається кожного разу перед відображенням чи оновленням (рендерингом) сцени;

`tock` – викликається після рендерингу сцени;

`play` – викликається кожного при поновленні рендерингу сцени;

`pause` – викликається при зупинці рендерингу сцени;

`updateSchema` – викликається кожен раз після оновлення схеми.

Схема описує властивості компонента та визначається у такий спосіб:

```
AFRAME.registerComponent('my-component', {
  schema: {
    arrayProperty: {type: 'array', default: []},
    integerProperty: {type: 'int', default: 5}
```

```

    }
}

```

У даному випадку компонент `my-component` буде містити дві властивості – `arrayProperty` та `integerProperty`. Щоб передати їх до компонента, потрібно задати значення відповідного атрибута.

```
<a-entity my-component="arrayProperty: 1,2,3;
integerProperty: 7"></a-entity>
```

Отримати ці властивості всередині компонента можна через властивість `data`. Отримати властивість `data` компонента із сущності, до якої він доданий, можна за допомогою `getAttribute`, а за допомогою `setAttribute` – встановити властивість у певне значення.

Системи у A-Frame задаються параметрами `a-scene` та реєструються через `AFRAME.registerSystem (name, definition)` – саме так бібліотекою Ar.js зареєстрована система arjs. На відміну від компонентів, системи надають лише методи `init, play, pause, tick та tock`.

A-Frame активно використовує об'єктну модель веб-браузера:

а) доступ до будь-якого об'єкту A-Frame може бути отриманий через `document.querySelector, document.getElementById` тощо;

б) різні компоненти можуть обмінюватись повідомленнями, для чого один з них повинен генерувати повідомлення за допомогою функцією `emit` (приймає три параметри: назву події; дані, які треба передати; ознака спливання (bubbling) події), а інший – обробляти («прослуховуючи» чергу подій) за допомогою методу, визначеного у `addEventListener`;

в) до об'єктів A-Frame можуть бути застосовані методи `setAttribute` (надання атрибуту значення), `removeAttribute` (видалення атрибуту), `createElement` (створення елементу) та `removeChild` (видалення).

Постійно поповнюваний перелік нових компонентів A-Frame, розроблених користувачами та оформленіх у пакети npm, доступний за посиланням <https://www.npmjs.com/search?q=aframe-component>

Для того, щоб застосувати компонент із списку, необхідно традиційно

перейти до його репозитарію, завантажити та підключити відповідний файл. Інший спосіб – скористатись сервісом unpkg, який надає можливість швидко завантажити будь-який файл з прт-пакету за посиланням вигляду

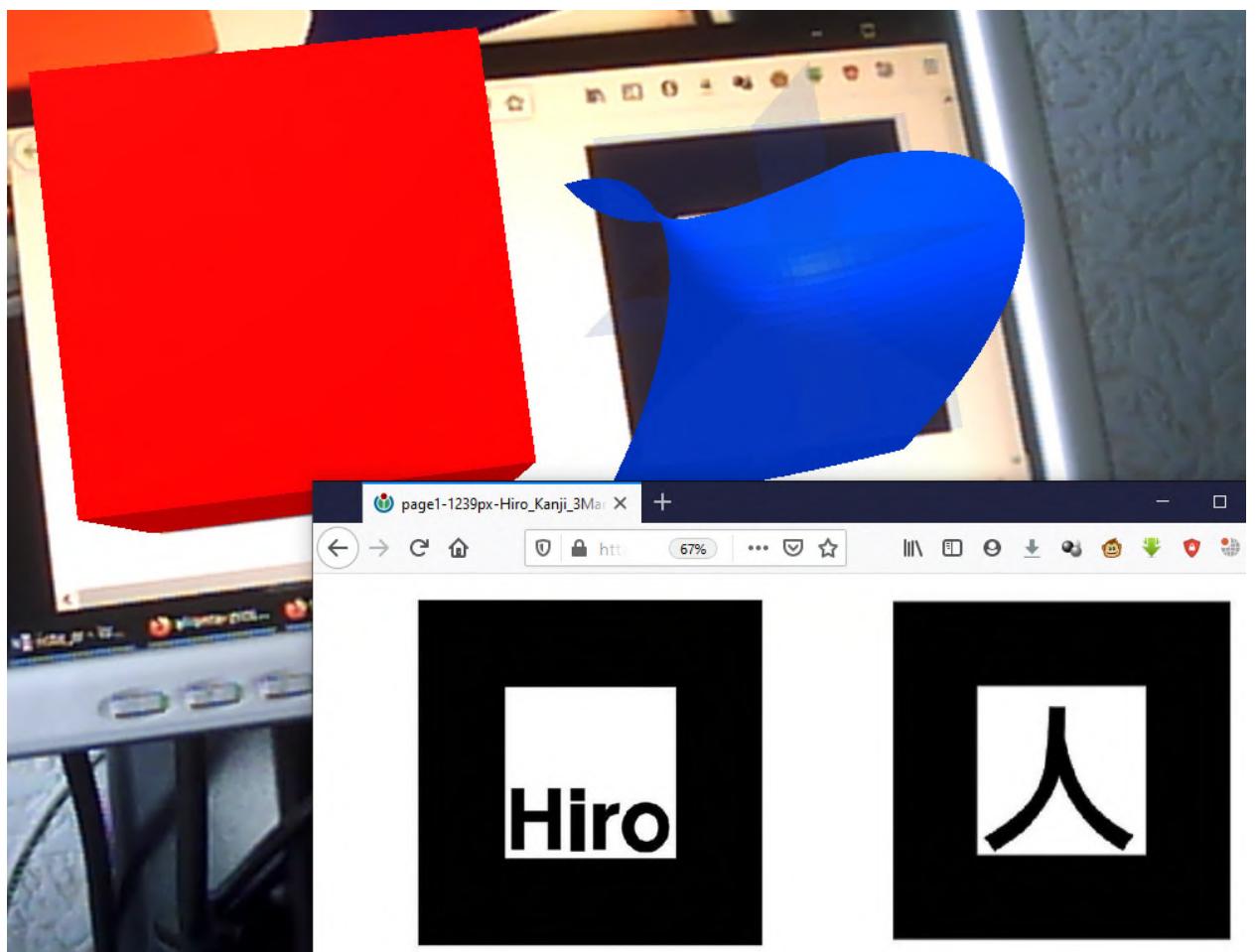
[unpkg.com/:package@:version/:file](https://unpkg.com/:package@:version/:file)

В якості прикладу скористаємося компонентом для побудови 3D-поверхонь:

```
<script src="https://unpkg.com/aframe-plot-component/dist/aframe-plot-component.min.js"> </script>
```

Для його використання замінимо блакитний куб, що пов'язаний із другим маркером, на поверхню:

```
<a-entity plot="function: ((3*x)^2 - (4*y)^2)/4;
order: 32;
show_zero_planes: true;
bounds: -0.5 0.5 -0.5 0.5 -0.5 0.5;
color: #04F"> </a-entity>
```

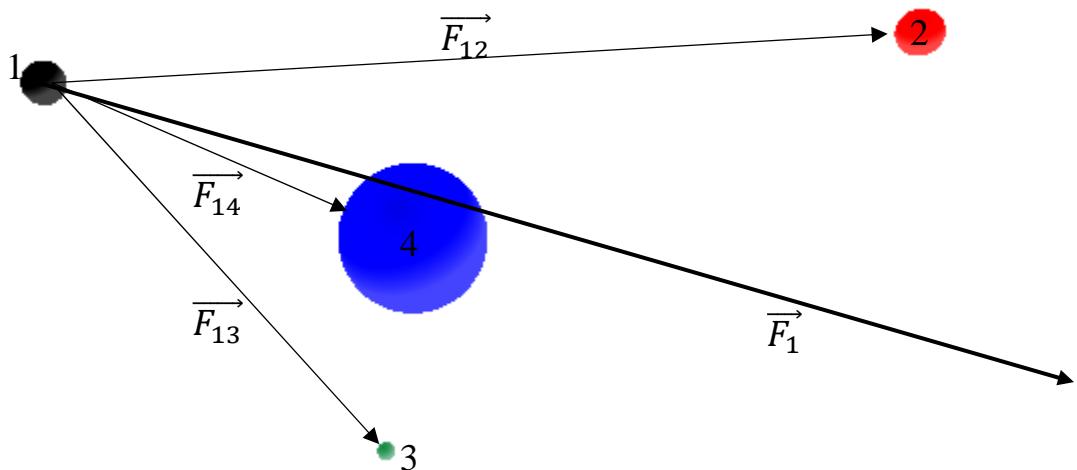


## 2.7. Приклад: модель Сонячної системи

Надамо об'єктам A-Frame фізичного змісту, поставивши задачу моделювання руху тіл в полі сил тяжіння. Для цього згадаємо два основних закони руху:

- 1) під дією сили  $\vec{F}$  тіло масою  $m$  набуває прискорення  $a$ ;
- 2) сила  $F$ , з якою притягуються два тіла, прямо пропорційна добутку їх мас  $m_1$  та  $m_2$  і обернено пропорційна квадрату відстані  $r$  між ними.

У першому випадку традиційно використовується векторний запис, у другому – скалярний. Приведемо запис до єдиної форми, уважаючи, що тіла рухаються у тривимірному просторі.



Прискорення  $\vec{a}_1$  набуває тіло масою  $m_1$  під дією сили  $\vec{F}_1$ , яка є сумою сил  $\vec{F}_{12}$ ,  $\vec{F}_{13}$  та  $\vec{F}_{14}$ . Кожна з них, у свою чергу, за законом всесвітнього тяжіння визначається формулою виду

$$F_{ij} = G \frac{m_i m_j}{r_{ij}^2}.$$

У загальному випадку сила  $\vec{F}_{ij}$  направлена від тіла масою  $m_i$  до тіла масою  $m_j$  уздовж вектору  $\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$ , що з'єднує центри їх мас ( $\vec{r}_i$  – координатний радіус-вектор тіла  $i$ ), тому попередню формулу краще переписати у такий спосіб:

$$\vec{F}_{ij} = G \frac{m_i m_j}{|\vec{r}_{ij}|^2} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|} = G \frac{m_i m_j}{|\vec{r}_{ij}|^3} \vec{r}_{ij}.$$

Тоді сила  $\vec{F}_i = m_i \vec{a}_i$ , що діє на тіло масою  $m_i$  з боку інших  $n$  тіл, може бути записана як їх сума:

$$\vec{F}_i = \sum_{\substack{j=1 \\ (i \neq j)}}^n \vec{F}_{ij} = \sum_{\substack{j=1 \\ (i \neq j)}}^n G \frac{m_i m_j}{|\vec{r}_{ij}|^3} \vec{r}_{ij} = G m_i \sum_{\substack{j=1 \\ (i \neq j)}}^n \frac{m_j \vec{r}_{ij}}{|\vec{r}_{ij}|^3},$$

або

$$m_i \vec{a}_i = G m_i \sum_{\substack{j=1 \\ (i \neq j)}}^n \frac{m_j \vec{r}_{ij}}{|\vec{r}_{ij}|^3},$$

звідки визначаємо прискорення  $\vec{a}_i$ , що набуває тіло  $i$  під дією інших:

$$\vec{a}_i = G \sum_{\substack{j=1 \\ (i \neq j)}}^n \frac{m_j \vec{r}_{ij}}{|\vec{r}_{ij}|^3}.$$

Ураховуючи, що  $\vec{a}_i = \frac{d\vec{v}_i}{dt}$ , а  $\vec{v}_i = \frac{d\vec{r}_i}{dt}$ , після переходу до скінченних різниць отримуємо наступну розрахункову схему:

- 1) визначити прискорення, яке набуває тіло  $i$  під дією з боку інших тіл;
- 2) визначити зміну швидкості  $i$ -го тіла:  $\Delta \vec{v}_i = \vec{a}_i \Delta t$ ;
- 3) визначити зміну координати  $i$ -го тіла:  $\Delta \vec{r}_i = \vec{v}_i \Delta t$ .

Її застосування потребує визначення початкових координат та швидкостей. Для цього скористаємося припущення про те, що усі тіла утворюють «парад планет», розташовуючись на вісі  $x$  та рухаючись вздовж перпендикулярної до неї вісі  $y$ .



Тоді початкова швидкість всіх тіл, за винятком першого (центрального), може бути визначена як

$$v_y = \frac{2\pi R}{T},$$

де  $R$  – середня відстань планети від Сонця, а  $T$  – період її обертання навколо Сонця.

Для зручності роботи спочатку налагодимо модель у віртуальній реальності на невеликій кількості об'єктів, узявши планети земної групи, а після того, як вона стане працездатною, додамо інші планети та перенесемо у доповнену реальність.

Після підключення бібліотеки A-Frame зареєструємо новий компонент – planet:

```
const day = 24.0*60*60; //тривалість земного дня у секундах
AFRAME.registerComponent('planet', {
  schema: {
    name: {type: 'string', default: ""}, //ім'я планети
    //середня відстань планети від Сонця
    dist: {type: 'number', default: 0},
    mass: {type: 'number', default: 0}, //маса планети, кг
    T: {type: 'int', default: 0}, //планетарний рік, земних днів
    v: {type: 'array', default: [0,0,0]}, //вектор швидкості
    a: {type: 'array', default: [0,0,0]}, //вектор прискорення
    //координатний радіус-вектор
    pos: {type: 'array', default: [0,0,0]}
  },
  init: function () {
    this.data.T*=day; //переводимо із земних днів у секунди
    this.data.pos[0]=this.data.dist; //розташовуємо на вісі x
    //візуальну позицію виражаємо у мільйонах кілометрів
    this.el.setAttribute('position',this.data.dist/1e9+' 0 0');
    if(this.data.T!=0)//для всіх об'єктів, крім Сонця,
      //обчислюємо початкову швидкість вздовж вісі y
    this.data.v[1] = 2*Math.PI*this.data.dist/this.data.T;
  }
});
```

```

    }
};


```

Метод `init` забезпечує правильне відображення об'єкту на сцені та додатково обчислює початкову швидкість об'єкта відносно центрального тіла, адже Сонце у даній моделі теж створюватиметься компонентом `planet`. Для того, щоб відрізнисти його від інших об'єктів, встановимо для нього значення параметру `t` у 0.

Новий компонент не визначає спосіб відображення об'єкту – для цього використовується відповідна сутність:

```

<a-scene>

  <a-sky color="black"></a-sky>

  <a-camera position="0 0 300" cursor-visible="true" cursor-
scale="2" cursor-color="#0095DD" cursor-opacity="0.5"></a-camera>

  <a-entity geometry="primitive: sphere; radius: 20.510"
material="color: yellow" planet="dist: 0; mass: 1.989e30; name:
Sun"></a-entity>

  <a-entity geometry="primitive: sphere; radius: 2.4397"
material="color: #AA5588" planet="dist: 57.910e9; mass: 3.285e23;
T: 88; name: Mercury"></a-entity>

  <a-entity geometry="primitive: sphere; radius: 6.0518"
material="color: white" planet="dist: 108.2e9; mass: 4.876e24; T:
224.7; name: Venus"></a-entity>

  <a-entity geometry="primitive: sphere; radius: 6.371"
material="color: cyan" planet="dist: 149.6e9; mass: 6e24; T: 365;
name: Earth"></a-entity>

  <a-entity main></a-entity>

</a-scene>
```

У створених сутностях можна легко змінити геометрію та матеріал об'єкту, і це не порушить реалізовану в компоненті логіку роботи. Зверніть увагу на розташування камери: вона відсунута по вісі  $z$  так, щоб можна було побачити орбіту поки що найвіддаленішої планети (подвоєна середня відстань Землі від Сонця) – при додаванні нових планет камеру доведеться відповідно відсувати далі.

За інтегрування рівнянь руху відповідатиме компонент main:

```
AFRAME.registerComponent('main', {
  init: function() {
    this.solar_system = document.querySelectorAll('[planet]');
  },
  tick: function (time, deltaTime) {
    const dt = day/3; //крок інтегрування
    const G=6.67e-11; //гравітаційна стала
    for(var i = 0; i<this.solar_system.length; i++) {
      planet_i=this.solar_system[i].getAttribute('planet');
      planet_i.a[0]=planet_i.a[1]=planet_i.a[2]=0;

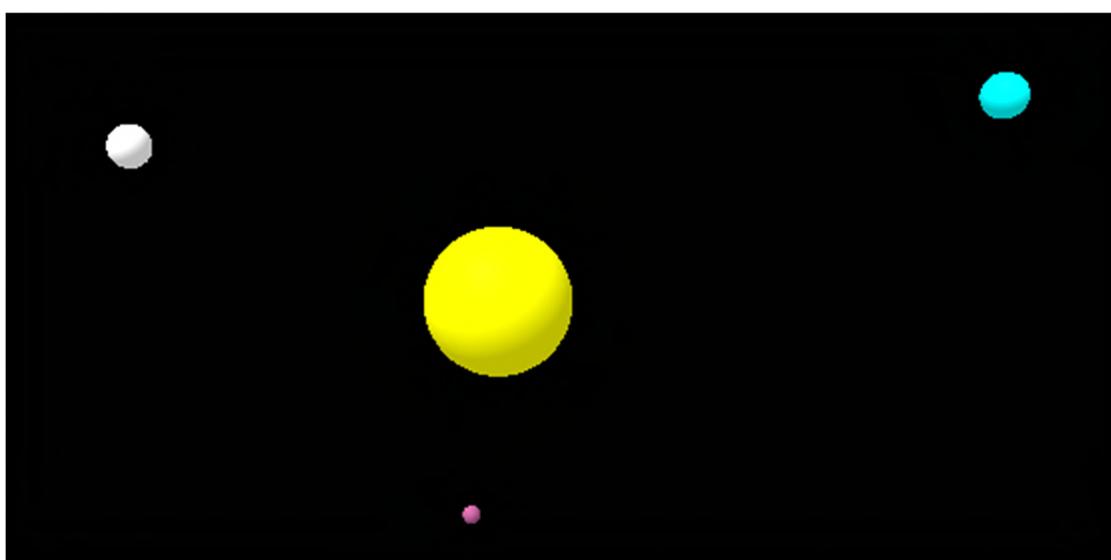
      for(var j = 0; j<this.solar_system.length; j++) {
        planet_j=this.solar_system[j].getAttribute('planet');
        if(i!=j) {
          deltapos = [0,0,0];
          for(var k = 0; k < 3; k++)
            deltapos[k]=planet_j.pos[k]-planet_i.pos[k];
          var r=Math.sqrt(Math.pow(deltapos[0],2) +
                         Math.pow(deltapos[1],2)+Math.pow(deltapos[2],2));
          for(var k = 0; k < 3; k++)
            planet_i.a[k]+=(G*planet_j.mass*deltapos[k]/
                           Math.pow(r, 3));
        }
      }
      for(var k = 0; k < 3; k++)
        planet_i.v[k]+=(planet_i.a[k]*dt);
      for(var k = 0; k < 3; k++)
        planet_i.pos[k]+=(planet_i.v[k]*dt);
      this.solar_system[i].setAttribute('position',
        (planet_i.pos[0]/1e9) +' '+ (planet_i.pos[1]/1e9) +
        ' '+ (planet_i.pos[2]/1e9));
    }
  }
})
```

```
});
```

Для того, щоб отримати всі визначені засобами A-Frame сутності з компонентом `planet`, у методі `init` компоненту `main` використовується метод `document.querySelectorAll`: отриманий масив об'єктів зберігається у змінній `solar_system`. Розмірність масиву  $n$  визначає кількість об'єктів гравітаційної взаємодії  $n$ . Кожен об'єкт масиву `solar_system` містить велику кількість елементів – для полегшення доступу до необхідних властивостей створених компонентів викликом `getAttribute` створюються дві змінні – `planet_i` та `planet_j`: просто запис із ними буде коротшим, ніж звернення до елементів `solar_system`. Ці та інші повторювані дії розміщені у методі `tick`, що відповідає за опрацювання події від таймеру.

Перш ніж розпочати обчислення прискорення, відповідні координатам  $x$ ,  $y$  та  $z$  елементи масиву  $a[0]$ ,  $a[1]$  та  $a[2]$  встановлюються у 0 (більш короткий спосіб показаний при визначенні `deltapos`). `deltapos` відповідає вектору  $\vec{r}_{ij}$  (відповідно `planet_j.pos - rj`, а `planet_i.pos - ri`), а  $r$  – його модулю. Цикли за змінною  $k$  проходять координати всіх векторів.

Обчислені координати об'єкту `pos` є масивом компоненту `planet`, у той час як атрибут `position` – рядком із `geometry`, що вимагає відповідного перетворення.



Додавання до сцени інших планет та компоненту `args` залишимо як завдання для самостійної роботи.

## 2.8. Застосування маркерів

Тег `<a-marker>` має такі основні атрибути:

`type` – тип маркеру: `'pattern'`, `'barcode'`, `'unknown'`;

`size` – розмір маркеру в метрах;

`url` – посилання на шаблон маркеру, якщо його тип – `'pattern'`;

`value` – значення коду, якщо тип маркеру – `'barcode'`;

`preset` – вибір стандартного маркеру (`'hiro'`, `'kanji'`);

`emitevents` – генерування подій `'markerFound'` та `'markerLost'`, якщо встановлений у `'true'`;

`smooth` – вмикає/вимикає згладжування зображення з камери (за замовчанням – `'false'`).

Тип `'barcode'` відповідає двовимірному (матричному) коду ARToolKit.

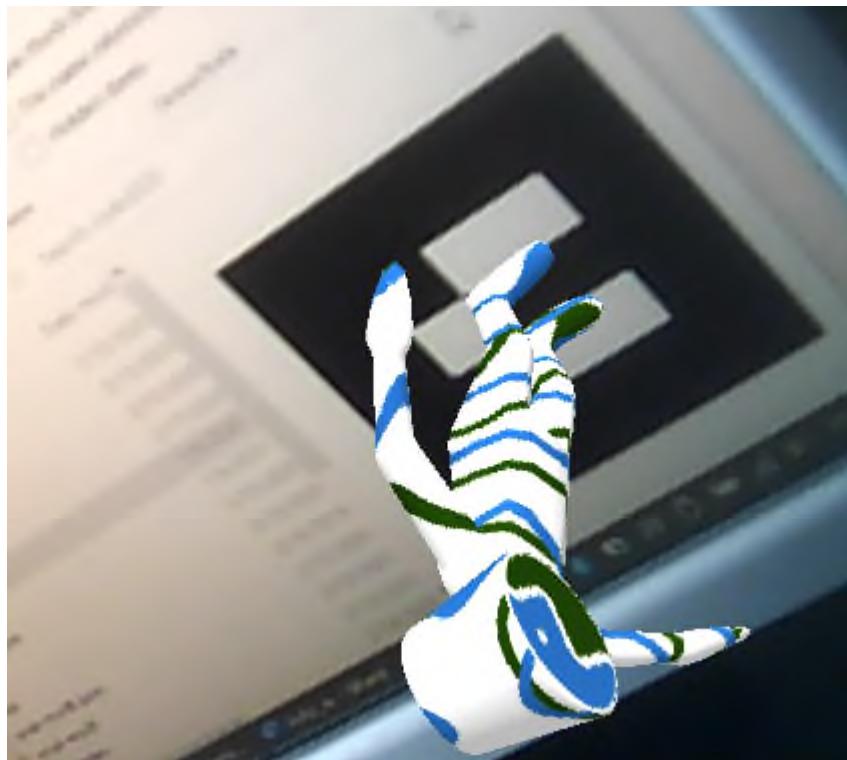
У репозитарії поточного розробника AR.js Ніколо Капріньолі за посиланням <https://github.com/nicolocarpignoli/artoolkit-barcode-markers-collection> можна знайти зображення для всіх типів матричних кодів, що задаються атрибутом `matrixCodeType` параметру `args` примітиву `a-scene`. Чим простіше матричний код, тим легше його розпізнати, але й тим менше маркерів можна створити за його допомогою: `3x3_HAMMING63` підтримує 8 маркерів (відстань Хеммінга – 3), `3x3_PARITY65` – 32 (1), `4x4_BCH_13_5_5` – 32 (5), `4x4_BCH_13_9_3` – 512 (3), `5x5_BCH_22_7_7` – 128 (7), `5x5_BCH_22_12_5` – 4096 (5). Чим більше відстань Хеммінга, тим краще маркер розпізнаватиметься.

Приклад налаштування сцени для використання матричних кодів:

```
<a-scene      vr-mode-ui="enabled: false"      embedded
args="debugUIEnabled:false;      detectionMode: mono_and_matrix;
matrixCodeType: 3x3;">
```

Відключення компонента `vr-mode-ui` надає можливість прибрати стандартні іконки A-Frame для переходу до повноекранних VR/AR режимів. Сам маркер створюється так:

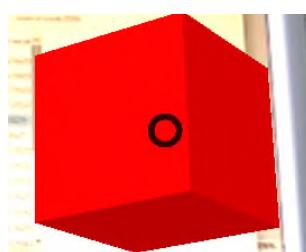
```
<a-marker type="barcode" value='7>' ... </a-marker>
```



Для опрацювання подій від клавіатури та миші необхідно вказати відповідний обробник події та підключити курсор до камери. У наступному прикладі показується, як можна анімувати об'єкт за подіями від миші, курсор якої має форму чорного кільця:

```
<a-scene vr-mode-ui="enabled: false" embedded arjs='sourceType: webcam; debugUIEnabled: false; detectionMode: mono_and_matrix; matrixCodeType: 3x3;'>

  <a-marker type='barcode' value='6'>
    <a-entity geometry="primitive: box" material="color: red" animation__mouseenter="property: rotation; from:0 0 0; to: 0 0 360; startEvents: mouseenter; dur: 1000"; animation__mouseleave="property: components.material.material.color; type: color; from: blue; to: red; startEvents: mouseleave; dur: 1000";> </a-entity>
  </a-marker>
  <a-entity camera><a-cursor></a-cursor></a-entity>
</a-scene>
```



Інший спосіб – явне використання обробників подій через виклик `addEventListener`. Наприклад, на рівні вікна веб-браузера можна визначити параметри підключеної камери під час її ініціалізації або подію припинення відеопотоку з камери:

```
window.addEventListener('camera-init', (data) => {
    console.log('camera-init', data);
})

window.addEventListener('camera-error', (error) => {
    console.log('camera-error', error);
})
```

Якщо необхідно опрацьовувати події, пов'язані із певною сущістю A-Frame, доцільно увести до неї новий компонент та зареєструвати його. Наприклад, компонент `registerevents`, доданий до маркера, допоможе розібратись, коли він потрапляє до поля зору камери (розвізнається), а коли – виходить з поля зору («губиться»):

```
AFRAME.registerComponent('registerevents', {
    init: function () {
        var marker = this.el;
        marker.addEventListener('markerFound', function() {
            console.log('markerFound', marker.id);
        });
        marker.addEventListener('markerLost', function() {
            console.log('markerLost', marker.id);
        });
    }
});
```

Приклад компонента для видимої сущності, що опрацьовує натискання кнопки миші на ньому:

```
AFRAME.registerComponent('jump', {
    init: function () {
        var obj = this.el;
        obj.addEventListener('click', function () {
            this.setAttribute('position', (Math.random() - 0.5) + " 0.5 0");
        });
    }
});
```

```
} );
```

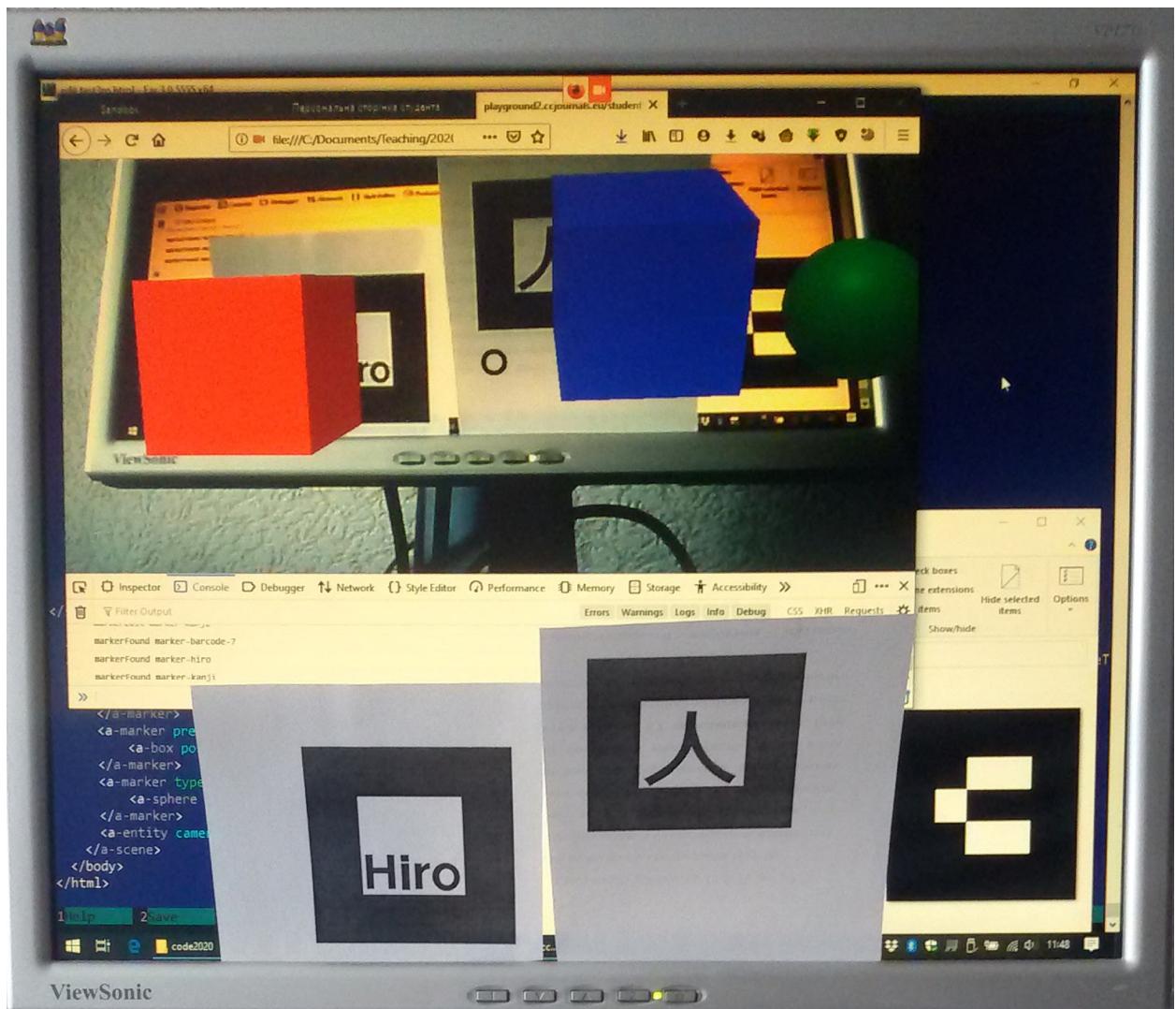
Приклад сцени з трьома незалежними маркерами, об'єкти яких реагують на натискання миші переміщенням:

```
<a-marker preset="hiro" id='marker-hiro' registerevents>
  <a-box position='0 0.5 0' material='color: red;' jump></a-box>
</a-marker>

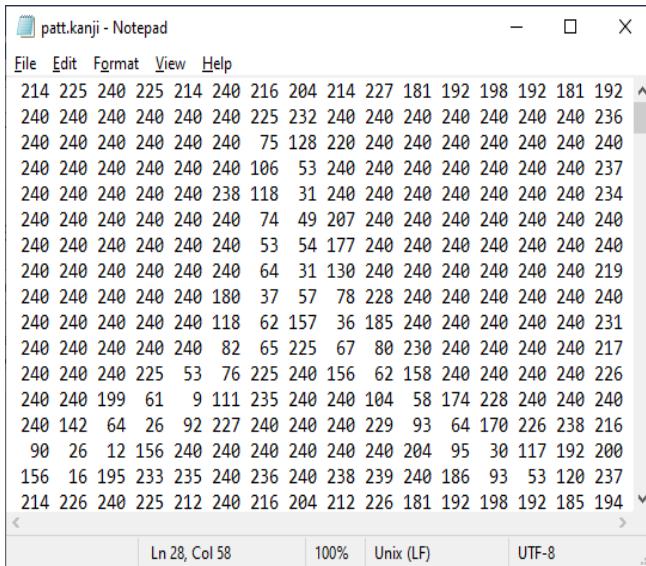
<a-marker preset="kanji" id='marker-kanji' registerevents>
  <a-box position='0 0.5 0' material='color:blue;' jump></a-box>
</a-marker>

<a-marker type='barcode' value='6' id='marker-barcode-7'
  registerevents>
  <a-sphere position='0 0.5 0' radius="0.5"
    material='color: green;' jump></a-sphere>
</a-marker>

<a-entity camera><a-cursor></a-cursor></a-entity>
```



AR.js надає можливість використання усіх стандартних маркерів, що підтримує ARToolKit. Крім вже використаних 'hiro' та 'kanji', це 'letterA', 'letterB', 'letterC', 'letterD', 'letterF', 'letterG' та ряд інших. Частина маркеру, що детектується – чорний квадрат зі стороною 1, усередині якого розміщено зображення. Опис розміщується у текстовому файлі (як правило, його ім'я звершується на .patt) – саме він використовується при розпізнаванні, а не звичне зображення.



The screenshot shows a Windows Notepad window titled "patt.kanji - Notepad". The menu bar includes File, Edit, Format, View, Help. The main text area contains a grid of binary values representing the marker pattern. The grid consists of 16 rows and 16 columns of 8-bit binary digits (0s and 1s). The bottom status bar shows Ln 28, Col 58, 100%, Unix (LF), and UTF-8.

```

patt.kanji - Notepad
File Edit Format View Help
214 225 240 225 214 240 216 204 214 227 181 192 198 192 181 192
240 240 240 240 240 240 225 232 240 240 240 240 240 240 240 236
240 240 240 240 240 240 75 128 220 240 240 240 240 240 240 240
240 240 240 240 240 240 106 53 240 240 240 240 240 240 240 237
240 240 240 240 240 238 118 31 240 240 240 240 240 240 240 234
240 240 240 240 240 240 74 49 207 240 240 240 240 240 240 240
240 240 240 240 240 240 53 54 177 240 240 240 240 240 240 240
240 240 240 240 240 240 64 31 130 240 240 240 240 240 240 219
240 240 240 240 240 180 37 57 78 228 240 240 240 240 240 240
240 240 240 240 118 62 157 36 185 240 240 240 240 240 240 231
240 240 240 240 82 65 225 67 80 230 240 240 240 240 240 217
240 240 240 225 53 76 225 240 156 62 158 240 240 240 240 226
240 240 199 61 9 111 235 240 240 104 58 174 228 240 240 240
240 142 64 26 92 227 240 240 229 93 64 170 226 238 216
90 26 12 156 240 240 240 240 204 95 30 117 192 200
156 16 195 233 235 240 236 240 238 239 240 186 93 53 120 237
214 226 240 225 212 240 216 204 212 226 181 192 198 192 185 194

```



Усього файл шаблону маркера зберігає 12 його зображень у 4 орієнтаціях. Створення маркеру, що гарно розпізнається – непроста задача: яким би не був розмір зображення, для використання ARToolKit воно буде закодоване усього 256 елементами (16 рядків на 16 стовпців), тому надмірно деталізоване зображення буде неминуче спрощене. Приклад гарного маркеру:



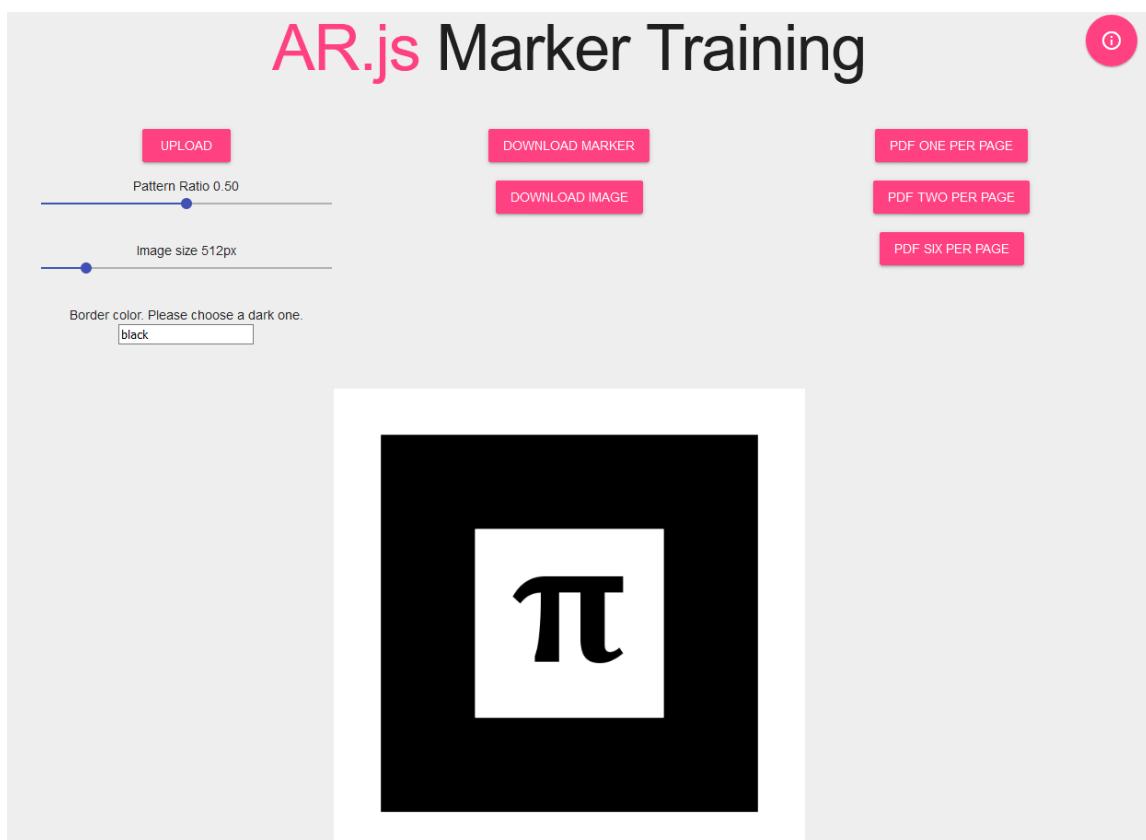
та його кодування (перше зображення):

```

255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 242 95 11 0 0 0 0 0 0 114 255 255 255
255 255 255 101 72 82 0 141 149 0 1 159 202 255 255 255
255 255 255 232 253 141 0 226 239 0 2 255 255 255 255 255
255 255 255 255 255 135 0 226 239 0 2 255 255 255 255 255
255 255 255 255 255 119 0 226 239 0 2 255 255 255 255 255
255 255 255 255 255 90 0 226 241 0 2 254 255 255 255 255
255 255 255 255 255 38 0 226 254 37 0 90 182 255 255 255
255 255 255 255 255 195 191 247 255 229 164 210 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255

```

Простий спосіб створити власний маркер – скористатись генератором маркерів <https://jeromeetienne.github.io/AR.js/three.js/examples/marker-training/examples/generator.html>



Отриманий текстовий файл із описом маркеру можна завантажити разом із його зображенням та використати для розпізнавання:

```
<a-marker type='pattern' url='pattern-pi.patt'>
...
</a-marker>
```

a-marker використовує статичну камеру, що розташована на початку координат та направлена у від'ємний бік вісі  $z$ . Видимість об'єктів, пов'язаних із маркером, встановлюється залежно від того, чи розпізнаний маркер камерою. Якщо один маркер використовується для того, щоб ініціювати певний процес, доцільно скористатись a-marker-camera. Після виходу маркеру з поля зору камери створені об'єкти прив'язуються до переміщення камери – саме за такого підходу модель Сонячної системи буде виглядати найкраще.

Для покращення розпізнавання доцільно використовувати асиметричні маркери, наприклад:



Ще один спосіб покращити розпізнавання – використання друкованих маркерів достатнього розміру. Вирізаючи маркер, залишайте біле поле ширину принаймні в половину чорної рамки – контраст є суттєвим при розпізнаванні. Якщо це не допомагає, можна збільшити роздільність камери у компоненті arjs – за замовчанням вона 640x480. Наприклад, для FullHD камери можна вказати arjs="sourceType: webcam; sourceWidth:1280; sourceHeight:960; displayWidth: 1280; displayHeight: 960; ".

У лютому 2020 року Ніколо Капріньолі анонсовано створення AR.js Studio, а також підтримка NFT (Natural Feature Tracking) – використання довільних зображень в якості маркерів (<https://carnaux.github.io/NFT-Marker-Creator/>) – у третій версії AR.js.

## 3 РОЗРОБКА ЗА ДОПОМОГОЮ THREE.JS

### 3.1. Приклад: побудова трикутника

A-Frame рекомендує не розміщувати «голий» JavaScript код, а прив'язувати його до компонентів задля дотримання архітектури ECS – саме так було зроблено у моделі Сонячної системи. Проте, проектуючи нові компоненти, зовсім не обов'язково застосовувати виключно A-Frame – можна послугуватись і тим, на чому він базується, аж до рівня WebGL. Розглянемо це на прикладі створення AR-програми, що передбачає одночасне використання кількох маркерів – побудову трикутника. Для цього розмістимо на сцені три сфери, прив'язані до маркерів із зображенням літерам А, В та С – шаблони для їх розпізнавання входять до складу AR.js:

```
<a-scene embedded vr-mode-ui="enabled: false;" arjs="debugUIEnabled: false; detectionMode: mono_and_matrix;">
  <a-marker type="pattern" url="https://raw.githubusercontent.com/jeromeetienne/AR.js/master/data/multimarkers/multi-abcdef/patt.a" id="A" registerevents>
    <a-sphere radius="0.10" color="red"></a-sphere>
    <a-entity id="lineAB"></a-entity>
  </a-marker>
  <a-marker type="pattern" url="https://raw.githubusercontent.com/jeromeetienne/AR.js/master/data/multimarkers/multi-abcdef/patt.b" id="B" registerevents>
    <a-sphere radius="0.10" color="red"></a-sphere>
    <a-entity id="lineBC"></a-entity>
  </a-marker>
  <a-marker type="pattern" url="https://raw.githubusercontent.com/jeromeetienne/AR.js/master/data/multimarkers/multi-abcdef/patt.c" id="C" registerevents>
    <a-sphere radius="0.10" color="red"></a-sphere>
    <a-entity id="lineAC"></a-entity>
  </a-marker>
<a-entity camera></a-entity>
```

```
<a-entity run></a-entity>
</a-scene>
```

Кожна із сфер є аналогом точки – вершини трикутника – та має ідентифікатор "A", "B" та "C" відповідно. Крім сфери, до маркеру прив'язані сутності без зовнішнього відображення – вони резервуватимуть місце для циліндрів – сторін трикутника – що з'єднуватимуть сфери. Сутність із компонентом `run` відповідатиме за логіку роботи програми, а компонент `registerevents` міститиме обробники подій «знаходження» та «втрати» маркера:

```
let markerVisible = { A: false, B: false, C: false };
AFRAME.registerComponent('registerevents', {
  init: function () {
    let marker = this.el;
    marker.addEventListener('markerFound', function() {
      markerVisible[ marker.id ] = true;
    });
    marker.addEventListener('markerLost', function() {
      markerVisible[ marker.id ] = false;
    });
  }
});
```

`markerVisible` – об'єкт JavaScript, що використовується для зберігання відомостей про те, які маркери в поточний момент є детектовані. Компонент `run` складають дві функції – початкова `init` та періодична `tick`:

```
AFRAME.registerComponent('run', {
  init: function() {
    this.A = document.querySelector("#A");
    this.B = document.querySelector("#B");
    this.C = document.querySelector("#C");
    this.p0 = new THREE.Vector3();
    this.p1 = new THREE.Vector3();
    this.p2 = new THREE.Vector3();
    let material = new THREE.MeshLambertMaterial(
      {color:0xFF0000});
  }
});
```

```

let geometry=new THREE.CylinderGeometry( 0.05, 0.05, 1, 12 );
geometry.applyMatrix( new THREE.Matrix4().makeTranslation(
0, 0.5, 0 ) );
geometry.applyMatrix( new THREE.Matrix4().makeRotationX(
THREE.Math.degToRad( 90 ) ) );
this.cylinderAB = new THREE.Mesh( geometry, material );
this.lineAB = document.querySelector('#lineAB').object3D;
this.lineAB.add( this.cylinderAB );
this.cylinderAB.visible = false;
this.cylinderBC = new THREE.Mesh( geometry, material );
this.lineBC = document.querySelector('#lineBC').object3D;
this.lineBC.add( this.cylinderBC );
this.cylinderBC.visible = false;
this.cylinderAC = new THREE.Mesh( geometry, material );
this.lineAC = document.querySelector('#lineAC').object3D;
this.lineAC.add( this.cylinderAC );
this.cylinderAC.visible = false;
} ,

tick: function (time, deltaTime) {
if ( markerVisible["A"] && markerVisible["B"] ) {
this.A.object3D.getWorldPosition(this.p0);
this.B.object3D.getWorldPosition(this.p1);
let distance = this.p0.distanceTo( this.p1 );
this.lineAB.lookAt( this.p1 );
this.cylinderAB.scale.set(1,1,distance);
this.cylinderAB.visible = true;
}
if ( markerVisible["B"] && markerVisible["C"] ) {
this.B.object3D.getWorldPosition(this.p1);
this.C.object3D.getWorldPosition(this.p2);
let distance = this.p1.distanceTo( this.p2 );
this.lineBC.lookAt( this.p2 );
this.cylinderBC.scale.set(1,1,distance);
this.cylinderBC.visible = true;
}
}

```

```

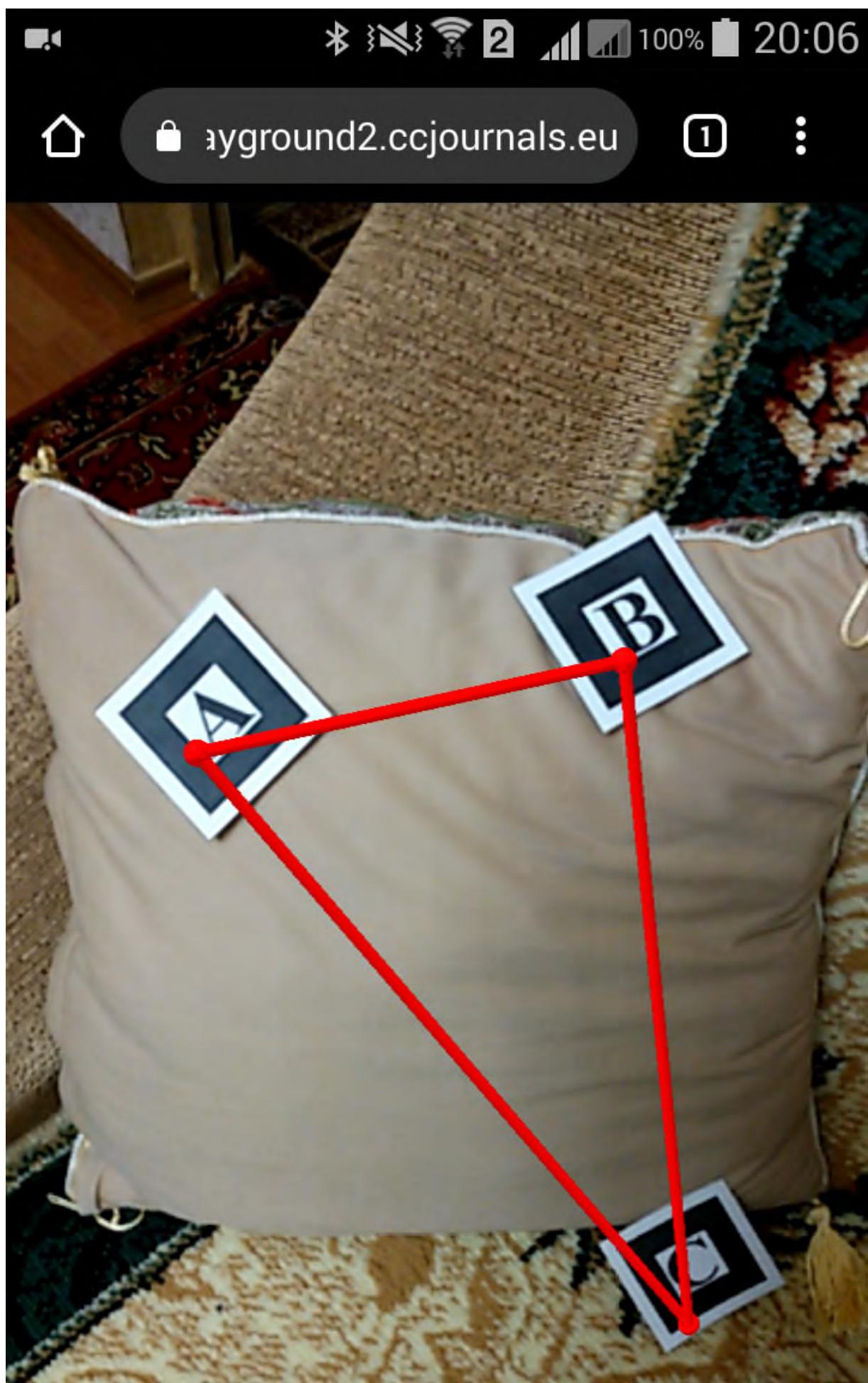
if ( markerVisible[ "A" ] && markerVisible[ "C" ] ) {
    this.A.object3D.getWorldPosition(this.p0);
    this.C.object3D.getWorldPosition(this.p2);
    let distance = this.p2.distanceTo( this.p0 );
    this.lineAC.lookAt( this.p0 );
    this.cylinderAC.scale.set(1,1,distance);
    this.cylinderAC.visible = true;
}
if ( !markerVisible[ "A" ] )
    this.cylinderAB.visible = this.cylinderAC.visible = false;
if ( !markerVisible[ "B" ] )
    this.cylinderAB.visible = this.cylinderBC.visible = false;
if ( !markerVisible[ "C" ] )
    this.cylinderAC.visible = this.cylinderBC.visible = false;
}
} );

```

На початку функції `init` у компоненті зберігаються посилання на вершини трикутника – сфери `this.A`, `this.B` та `this.C` – ізасобами бібліотеки `Three.js` створюються три точки – координатні вектори `p0`, `p1` та `p2`. Далі засобами тієї ж бібліотеки створюється циліндричні об'єкти, що прикріплюються до раніше створених сущностей `lineAB`, `lineBC` та `lineAC`.

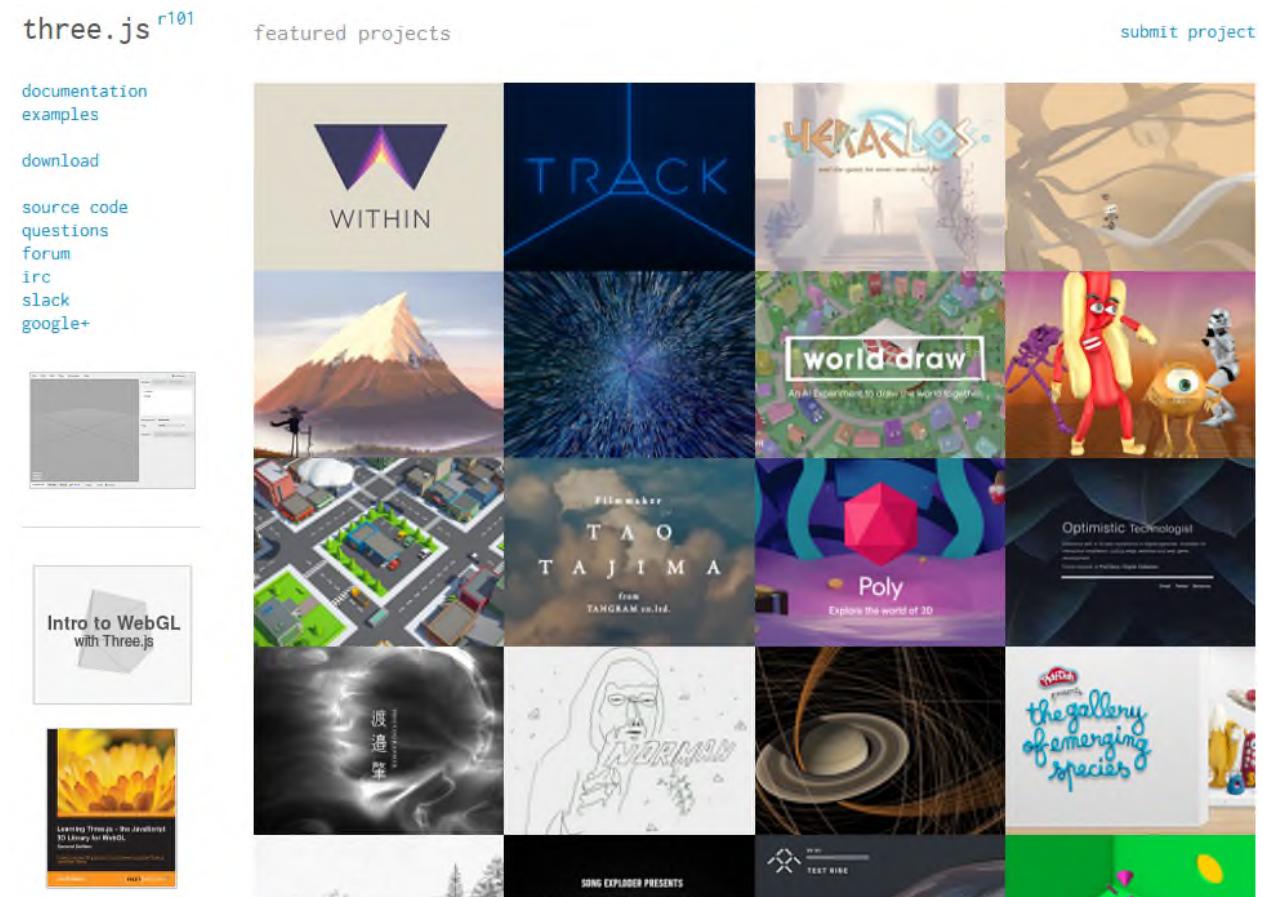
Функція `tick` регулярно аналізує, чи є маркери ідентифіковані попарно та поодинці. Для того, щоб побудувати трикутник, необхідно побудувати його сторони – відрізки. Відповідно, для побудови відрізку необхідні принаймні дві точки, що задають його початок та кінець: після визначення координат цих точок розраховується відстань між ними, далі циліндричний об'єкт спрямовується з до кінцевої точки і подовжується до неї. Коли маркер точки «зникає», пов'язані з нею відрізки приховуються встановленням атрибуту `visible` у `false`.

Наведений код використовує ряд об'єктів та методів з бібліотеки, яка була використана при розробці A-Frame – `Three.js`. Розглянемо більш детально, як можна розширити функціональність наших AR-програм з її використанням.



### 3.2. Обговорення Three.js та налаштування проекту

Three.js – це відкрита 3D-графічна бібліотека, написана на JavaScript. На сайті <https://threejs.org/> можна знайти велику кількість якісних демонстрацій, створених із її використанням.



Її автор Рікардо Мігель Кабелло, також відомий як mrdoob, є одним із пionерів використання WebGL, тому ця бібліотека часто використовується при побудові інших бібліотек – так, у каталозі з попереднього розділу AR.js-master можна побачити каталог із нею three.js.

За посиланням «source code» на сайті Three.js можна перейти до репозитарію GitHub та клонувати ZIP-файл із бібліотекою.

Для подальшої роботи створимо каталог threeJS, до якого розпакуємо вміст архіву, та розмістимо індексний файл threeJS.html, що міститиме стандартні теги типу документу, початку документу HTML, заголовок та тіло.

До тіла документа додамо тег canvas, що з'явився у HTML5 та призначений для створення графіки засобами JavaScript:

JavaScript 3D library. <https://threejs.org/>

javascript 3d scene-graph html5 svg canvas webgl webvr

26,501 commits 4 branches 93 releases 1,041 contributors MIT

Branch: dev New pull request Find file Clone or download

mrdoob Merge pull request #15826 from Mugen87/dev24 ...

- .github r101
- build Updated builds.
- docs Add a collada exporter link to the docs
- editor Only display tenions of catmullrom is selected

Clone with HTTPS Use Git or checkout with SVN using the web URL.  
https://github.com/mrdoob/three.js.git

Open in Desktop Download ZIP

За замовчанням canvas створює полотно з ім'ям "Canvas", по замовчанию ширину 300 пікселів, а висоту – 150.

Далі виконаємо підключення бібліотеки Three.js:

```
<script src="three.js-dev/build/three.js"></script>
```

Отриманий файл є основою для подальших модифікацій.

```
<!DOCTYPE html>
<html>
    <head>
        <title>threeJS</title>
    </head>
    <body>
        <canvas id="Canvas"></canvas>
        <script src="three.js-dev/build/three.js"></script>

    </body>
</html>
```

Для того, щоб перевірити його працездатність на віддаленому сервері, необхідно скопіювати на нього не лише індексний файл threeJS.html, а й увесь чималенький каталог three.js-dev, що може зайняти певний час. Його левову частину складають приклади (three.js-dev/examples) та документація (three.js-dev/docs).

Віддалений доступ до файлу threeJS.html можливий за прямим посиланням (достатньо вказати повний шлях до нього на сервері), проте більш правильним буде додати посилання на цей файл до індексного файлу серверу:

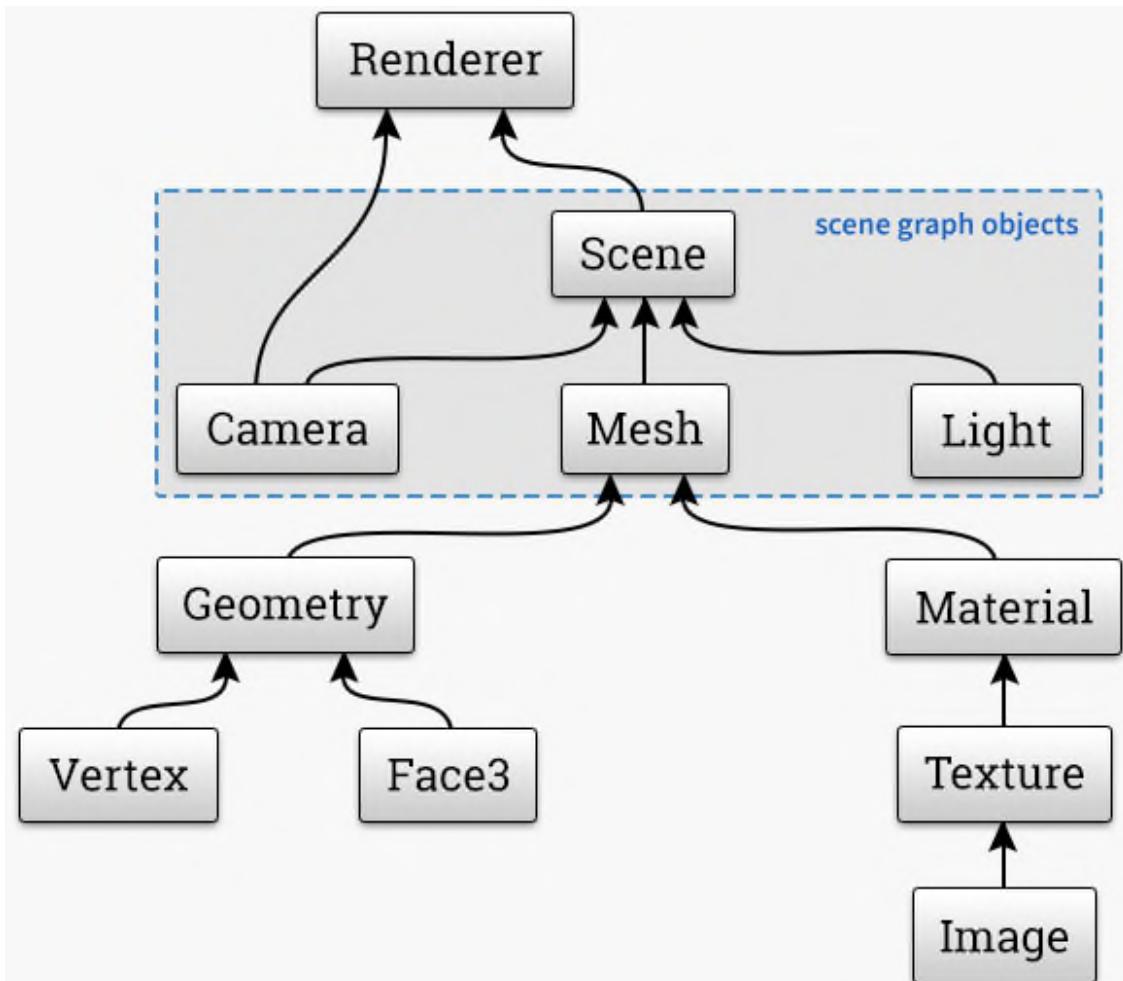
[<a href="threeJS.html" target="\\_blank">](threeJS.html)

Перший приклад використання threeJS</a>



### 3.3. Створення базової сцени

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.):

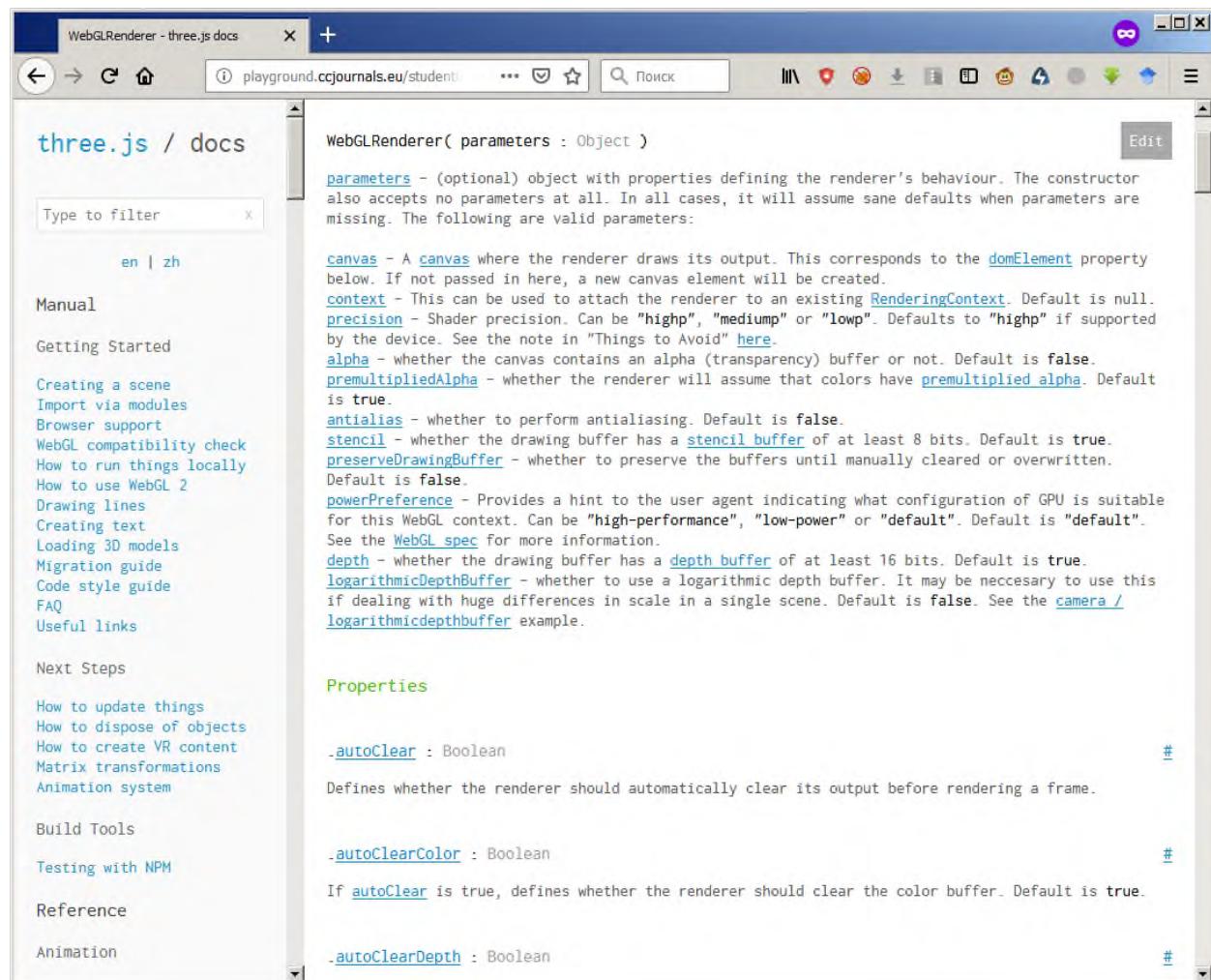


Нами було використано тег canvas, тому всі дії із розміщення об'єктів на створеному полотні будуть виконуватись мовою JavaScript між відповідними тегами <script> та </script>.

Перше, що необхідно створити, це рендерер – те, що буде відображати 3D-модель на полотні з урахуванням матеріалу, текстури та освітлення:

```
var renderer=new THREE.WebGLRenderer({canvas: document.getElementById('Canvas'), antialias: true});
```

Зміна 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.



Налаштовуються два з параметрів – canvas (полотно) та antialias (згладжування). Ураховуючи, що полотно було створено раніше за допомогою <canvas id="Canvas"></canvas>, для доступу до нього виконується виклик методу getElementById об'єкту document – складової DOM – з ідентифікатором полотна Canvas. Згладжування («антиаліасінг») надає

можливість зробити прибрати «зубці», що виникають на краях об'єктів, особливо коли їх багато.

Повний перелік властивостей та методів класу `WebGLRenderer` доступні у документації. Проілюструємо деякі з них:

`renderer.setClearColor(0xCBEFFF);` – встановлює фоновий колір, близький до блакитного;

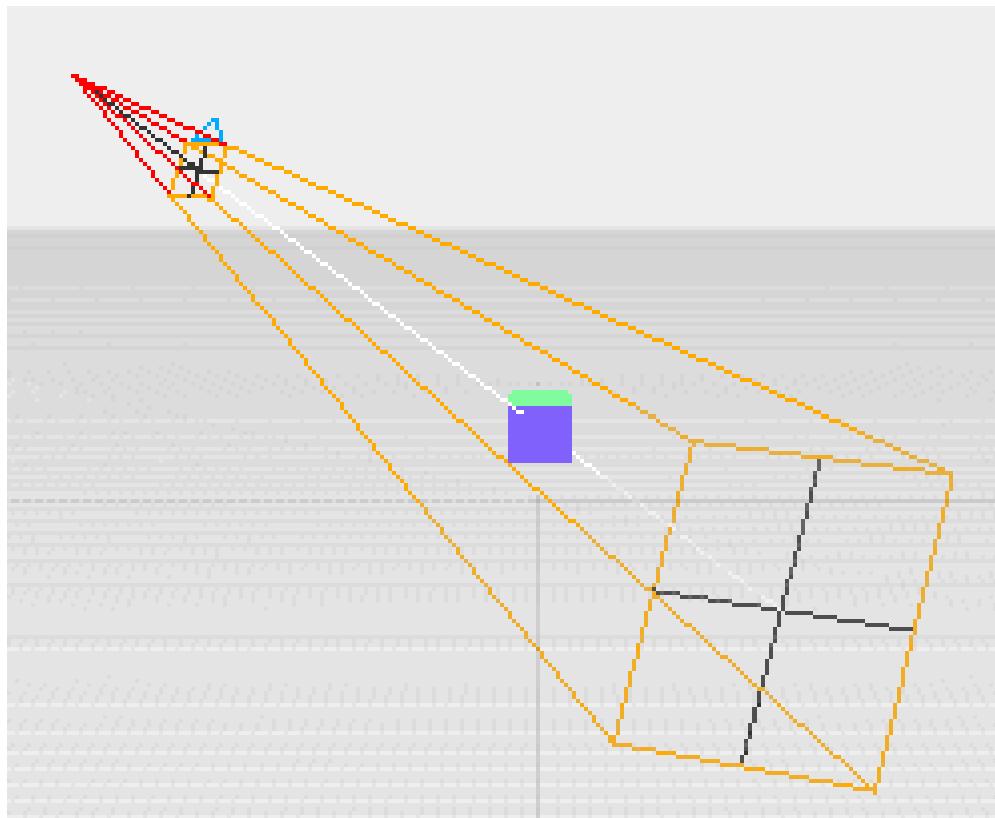
`renderer.setPixelRatio(window.devicePixelRatio);` – встановлює співвідношення пікселів у те саме значення, що й у вікні браузера;

`renderer.setSize(window.innerWidth, window.innerHeight);` – розширяє полотно до висоту та ширину поточного вікна.

Наступний об'єкт, що створюється – перспективна камера:

```
var camera=new THREE.PerspectiveCamera(35, window.innerWidth / window.innerHeight, 0.1, 3000);
```

Параметрами конструктора є складові зірзаної піраміди огляду: перший – вертикальне поле зору, другий – співвідношення сторін, третій – найближча площа, четвертий – найдальша.



Для створення сцени необхідно викликати конструктор без параметрів `Scene`:

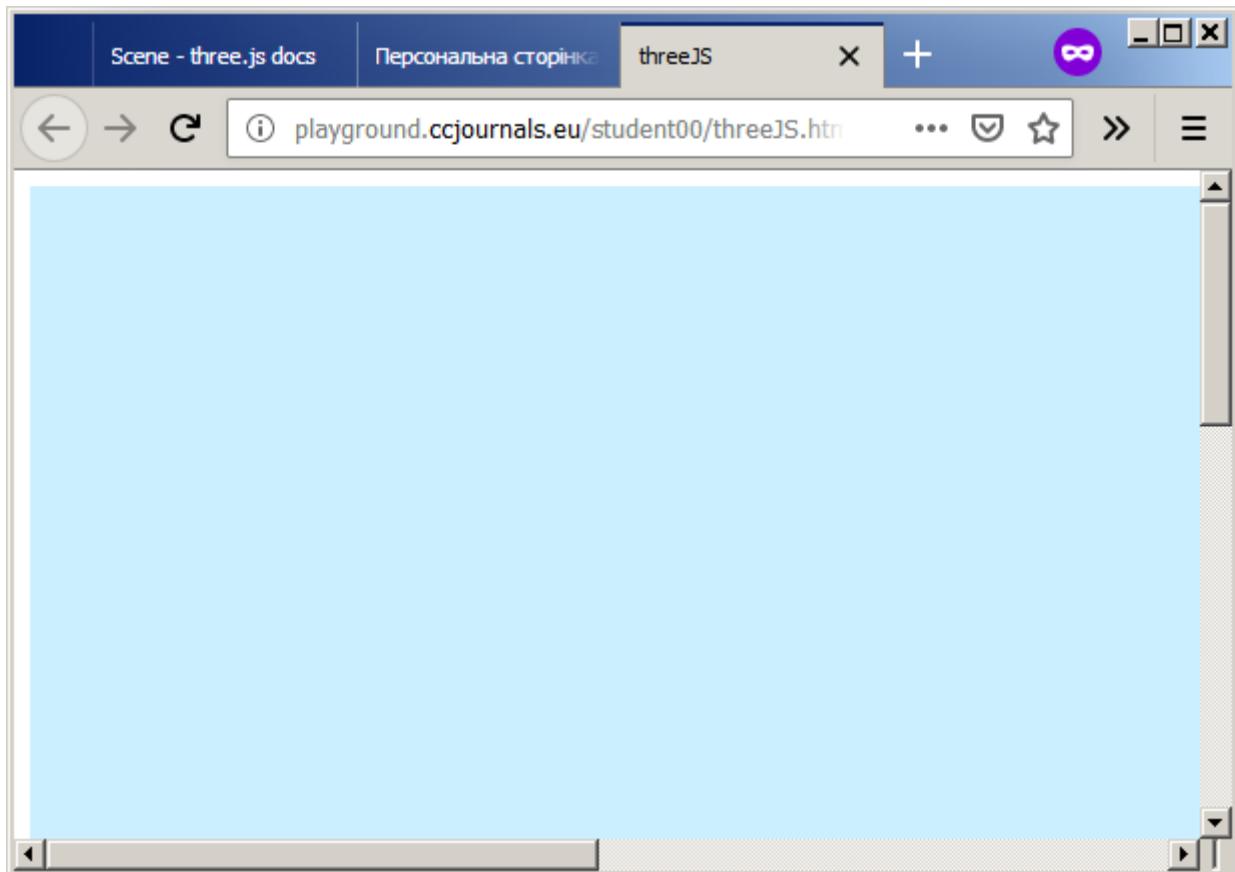
```
var scene=new THREE.Scene();
```

Для того, щоб рендерер, камера та сцена запрацювали, необхідно створити функцію для їх візуалізації. У цій функції спочатку з об'єкту renderer викликається метод render, параметрами якого є об'єкти scene та camera, а далі викликом requestAnimationFrame встановлюється, що дана функція її буде виконувати рендеринг. Після визначення функції її слід викликати – саме це є розпочненням процесу анімації:

```
function render()
{
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}

render();
```

За умови правильного налаштування вікна виглядатиме так:



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

- 1) визначення геометрії об'єкту – векторів позиції, кольорів та ін.;
- 2) визначення матеріалу – способу рендерингу об'єкту;

3) композиція геометрії та матеріалу.

Так, для створення піраміди:

1) скористаємось конструктором класу CylinderGeometry:

```
var pyramidgeometry=new THREE.CylinderGeometry(0, 0.8, 2, 4);
```

2) визначимо матеріал, що відбиває промені:

```
var pyramidmaterial=new THREE.MeshLambertMaterial(  
    {color: 0xF3FFE2});
```

3) комбінуємо геометрію та матеріал:

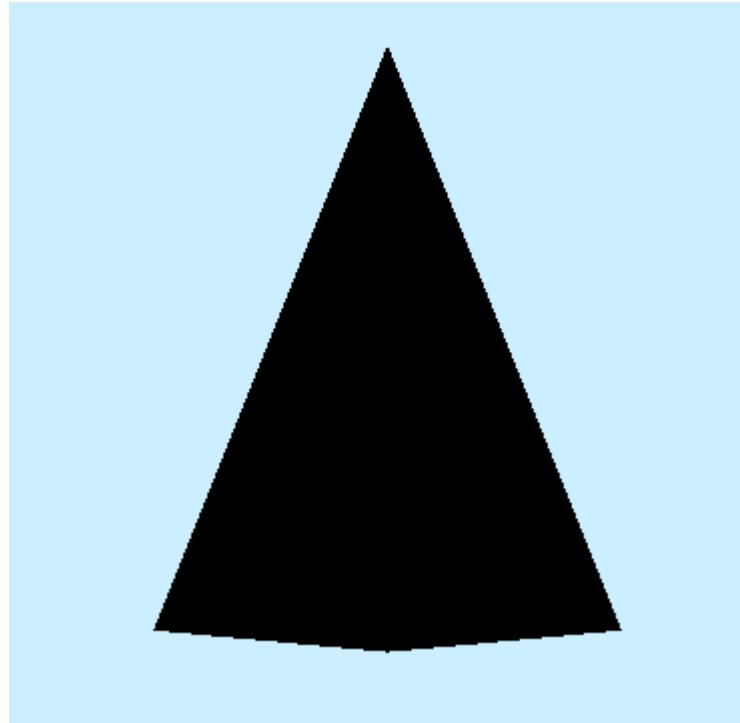
```
var pyramidmesh=new THREE.Mesh(pyramidgeometry, pyramidmaterial);
```

Створена у такий спосіб піраміда буде розташована у початку координат. Для зміни її позиції скористаємось властивістю position, успадкованою класом Mesh від свого батька – Object3D. Данна властивість є об'єктом класу Vector3, а set – його методом:

```
pyramidmesh.position.set(0, 2, -10);
```

Останній крок – розміщення об'єкту на сцені:

```
scene.add(pyramidmesh);
```



Документація Three.js містить інтерактивні демонстрації, що надають можливість переглянути різні об'єкти та модифікувати їх параметри.

The screenshot shows the Three.js documentation for the `CylinderGeometry` class. On the left, there's a sidebar with links to various sections like Manual, Getting Started, and Reference. The main content area has a title "CylinderGeometry" and a subtitle "A class for generating cylinder geometries". Below the title is a 3D preview of a cylinder with a polygonal base. To the right of the preview is a control panel titled "THREE.CylinderGeometry" with sliders for radiusTop (7.8), radiusBottom (5), height (10), radialSegments (8), heightSegments (1), openEnded (checked), thetaStart (0), and thetaLength (6.3). At the bottom of the preview area is a link "Open in New Window". Below the preview, there's an "Example" section with sample code:

```
var geometry = new THREE.CylinderGeometry( 5, 5, 20, 32 );
var material = new THREE.MeshBasicMaterial( {color: 0xffff00} );
var cylinder = new THREE.Mesh( geometry, material );
scene.add( cylinder );
```

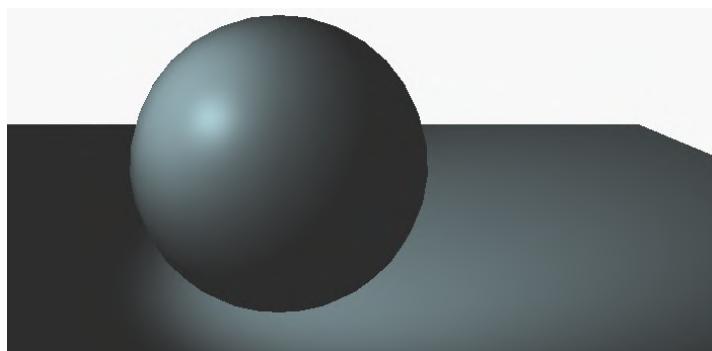
Underneath the example is a "Constructor" section with the following code signature:

```
CylinderGeometry(radiusTop : Float, radiusBottom : Float, height : Float, radialSegments : Integer, heightSegments : Integer, openEnded : Boolean, thetaStart : Float, thetaLength : Float)
```

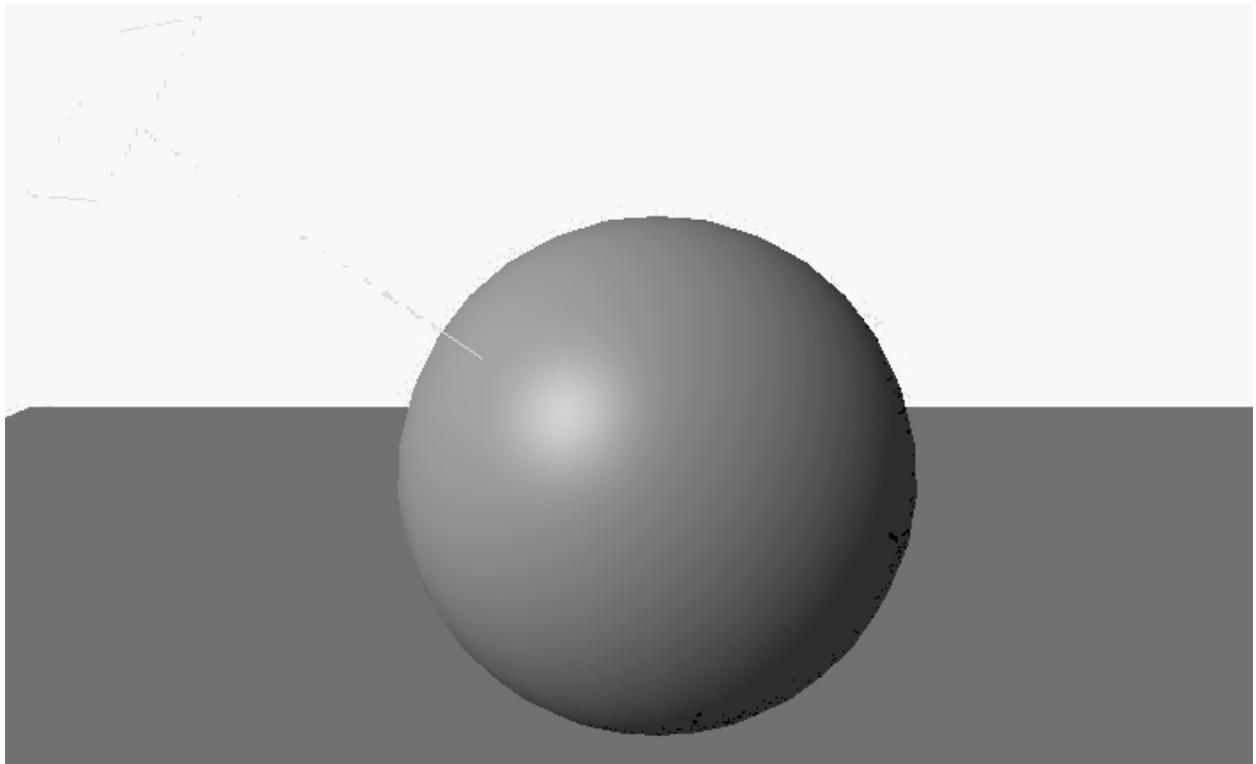
Автор AR.js Жером Етьєнн створив розширення для Chrome, що надає можливість налагоджувати програми із використанням Three.js – Three.js Inspector (<https://libraries.io/github/mrdoob/threejs-inspector>). Рікардо Кабелло надає можливість конструювання сцени за допомогою візуального редактора за посиланням <https://threejs.org/editor/> – так, якщо створити описаний вище об'єкт за його допомогою, він теж виглядатиме чорним.

І не дивно – для того, щоб `MeshLambertMaterial` сяяв, відбиваючи світло, необхідно додати його джерело. Three.js пропонує такі основні джерела світла:

`AmbientLight` – всі об'єкти сцени освітлюються рівномірно;

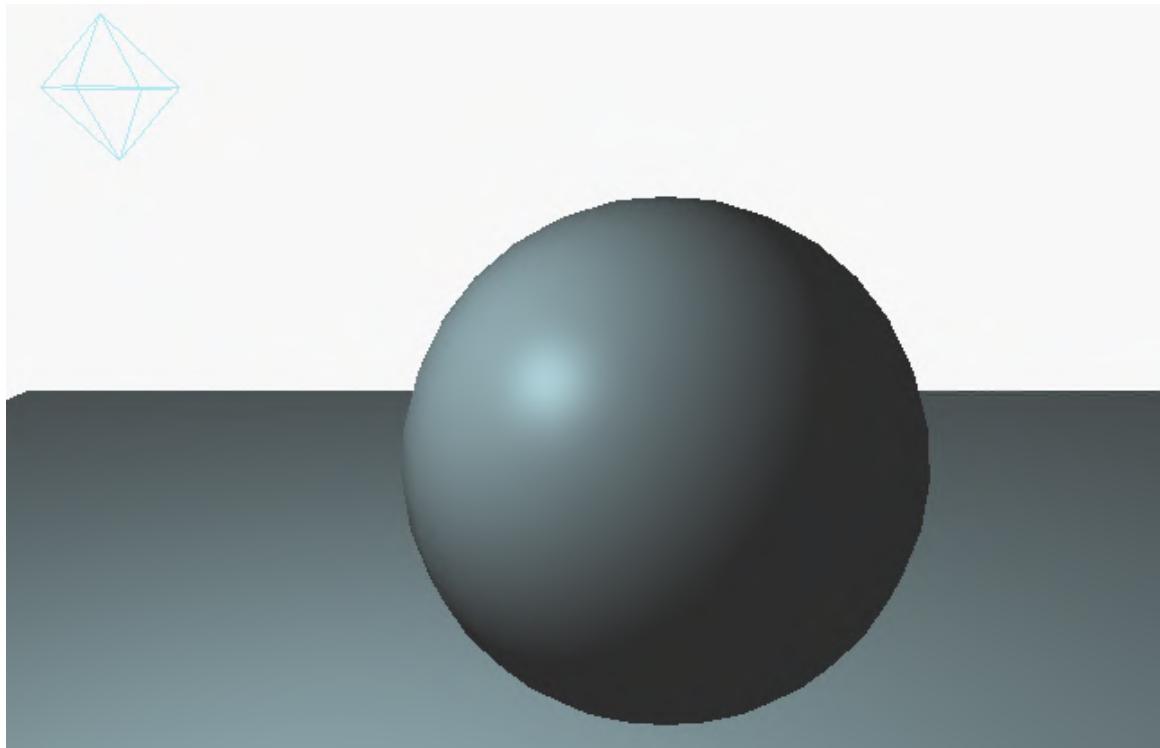


DirectionalLight – всі промені йдуть паралельно у вказано напрямі (денне світло);



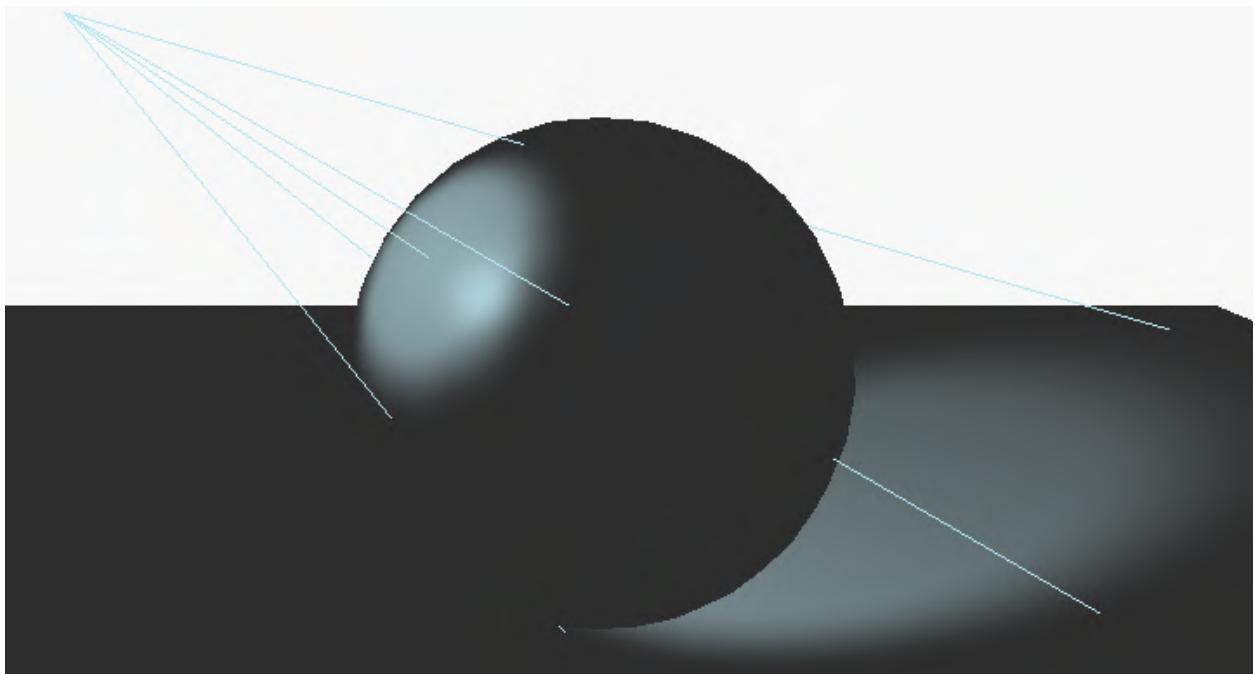
HemisphereLight – джерело світла знаходиться над сценою, при затіненні кольори змінюються від кольору неба до кольору землі;

PointLight – точкове джерело, промені якого поширюються в усіх напрямах (лампочка);



`RectAreaLight` – світло поширюється через прямокутник (яскраве вікно, відкриті освітлені двері);

`SpotLight` – світло поширюється з точки в одному напрямі вздовж конуса (прожектор).



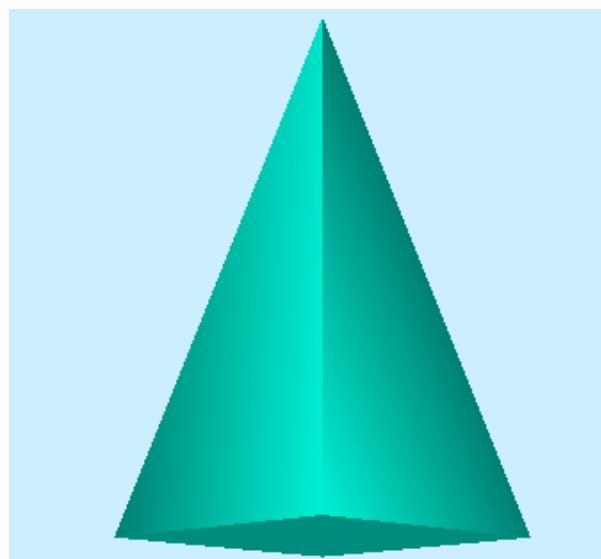
Додамо два джерела світла:

– розсіяне

```
var lightOne=new THREE.AmbientLight(0xfffff, 0.5);
scene.add(lightOne);
```

– точкове:

```
var lightTwo=new THREE.PointLight(0xfffff, 0.5);
scene.add(lightTwo);
```



Тепер можна побачити, що матеріал піраміди дійсно гарно відбиває світло – як навколошнє, так й точкове.

Отримане зображення виглядає статичним, хоча воно є анімованим – кожного разу, коли викликається функція рендерингу `render` (а вона викликається постійно), можна змінювати будь-які атрибути об'єкту. Так, якщо на початку функції вставити рядок

```
pyramidmesh.rotation.y+=0.1;
```

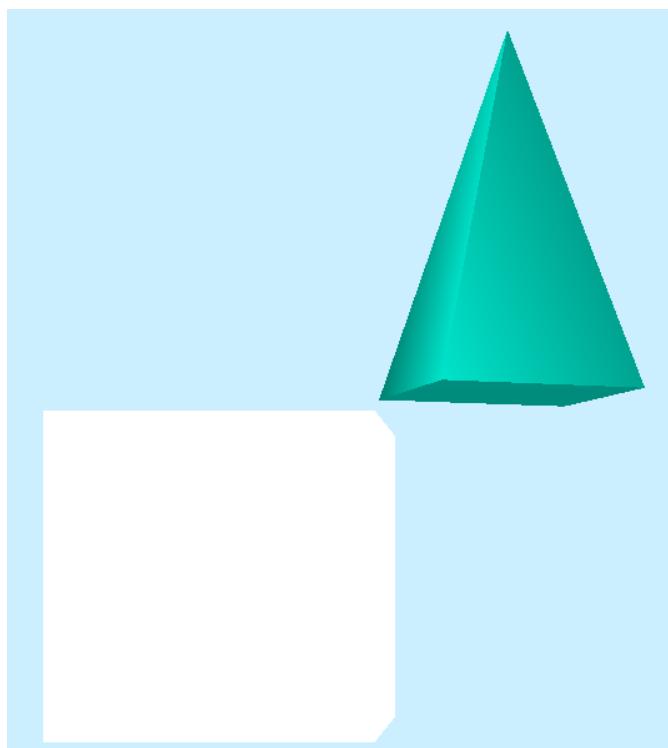
то ефектом від його виконання буде нескінченне обертання піраміди навколо її вертикальної вісі.

### 3.4. Дослідження різних геометрій

Поекспериментуємо з деякими іншими об'єктами, додаючи їх між пірамідою та джерелами світла.

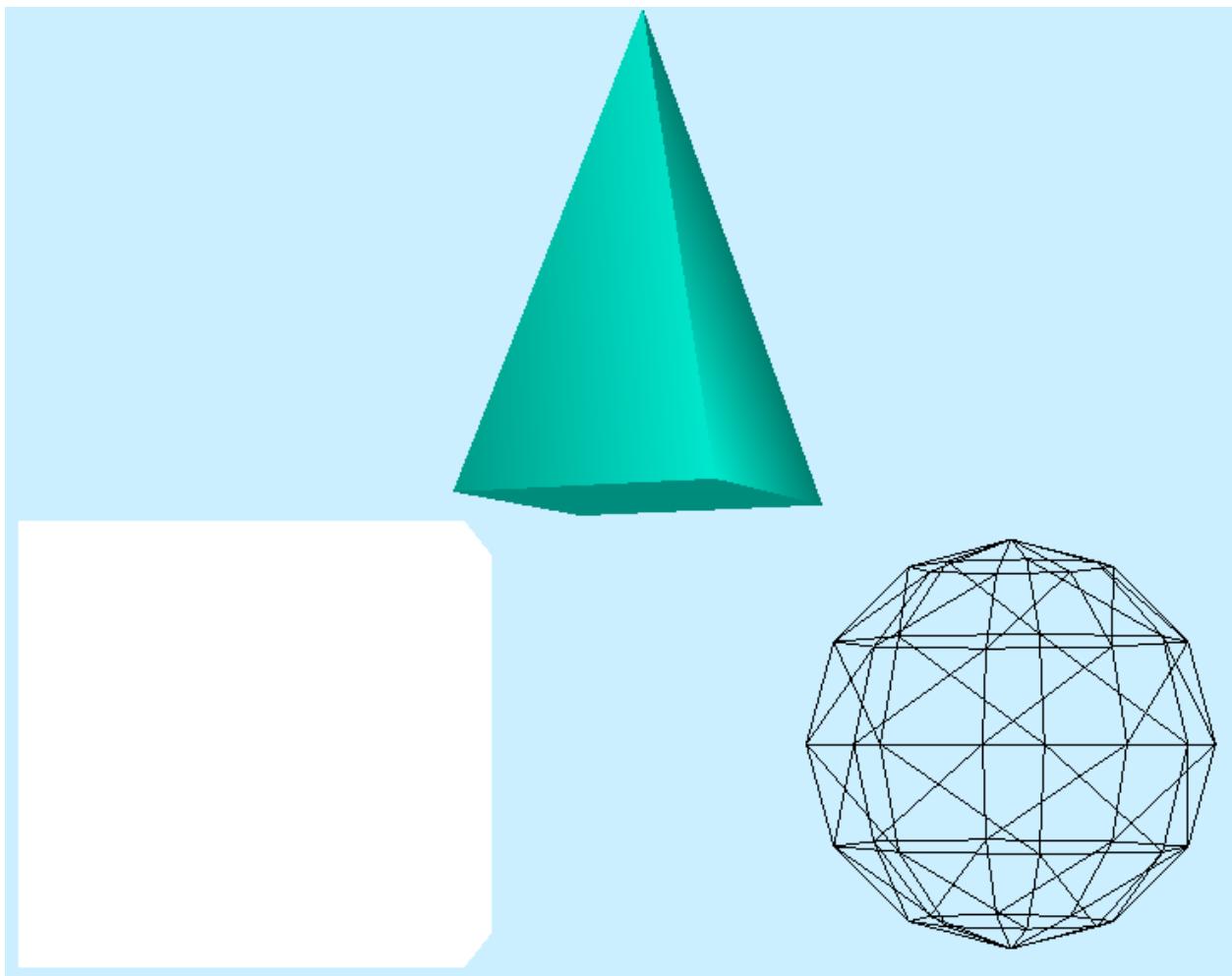
```
var boxgeometry=new THREE.BoxGeometry(1, 1, 1);
var boxmaterial=new THREE.MeshBasicMaterial();
var boxmesh=new THREE.Mesh(boxgeometry, boxmaterial);
boxmesh.position.set(-0.9, 0, -6);
scene.add(boxmesh);
```

За замовчанням, колір основного матеріалу – білий:



При створенні наступного об'єкту змінимо параметри основного матеріалу, обравши каркасне відображення лініями чорного кольору:

```
var spheregeometry=new THREE.SphereGeometry(0.5);
var spherematerial=new THREE.MeshBasicMaterial({wireframe: true,
                                                 color: 0x000000});
var spheremesh=new THREE.Mesh(spheregeometry, spherematerial);
spheremesh.position.set(0.9, 0, -6);
scene.add(spheremesh);
```



Three.js надає можливість використання двох основних типів геометрій: буферної (`BufferGeometry`) та звичайної (`Geometry`). Буферна геометрія ефективно подає сітки, лінії та точки в буферах, зменшуючи витрати на передачу всіх цих даних до графічного процесора, звичайна є менш ефективною, проте більш легкою для початківця. У зв'язку з цим більшість геометричних об'єктів має дві версії – буферну та звичайну:

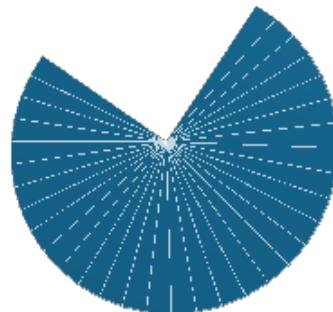
`PolyhedronBufferGeometry`,    `PolyhedronGeometry` – класи

багатогранників;

`BoxBufferGeometry`, `BoxGeometry` – класи для створення прямокутних паралепіпедів заданої ширини, висоти та глибини;



`CircleBufferGeometry`, `CircleGeometry` – круг, складений із ряду трикутних сегментів, які орієнтовані навколо центральної точки та можуть також використовуватися для створення правильних багатокутників;



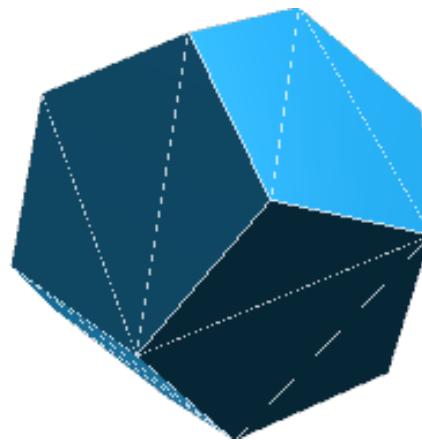
`ConeBufferGeometry`, `ConeGeometry` – класи конічних об'єктів;



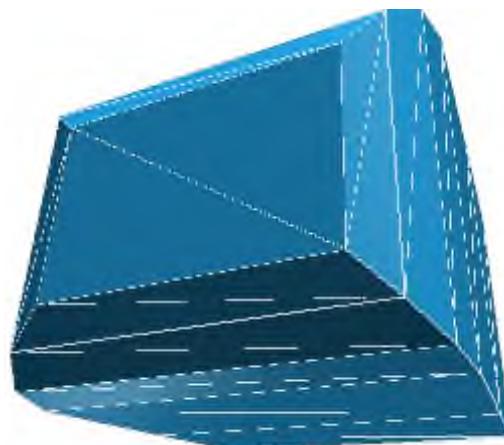
`CylinderBufferGeometry`, `CylinderGeometry` – класи циліндричних об'єктів;



DodecahedronBufferGeometry, DodecahedronGeometry – класи правильних дванадцятигранників;



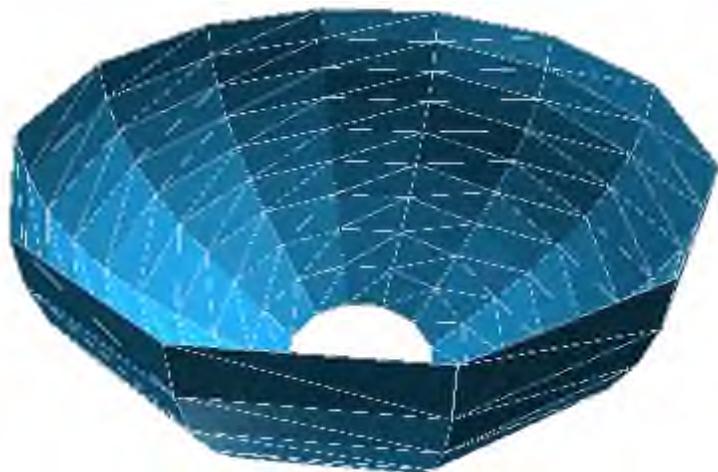
ExtrudeBufferGeometry, ExtrudeGeometry – класи для створення об'єктів за заданим набором точок;



IcosahedronBufferGeometry, IcosahedronGeometry – класи правильних двадцятигранників;



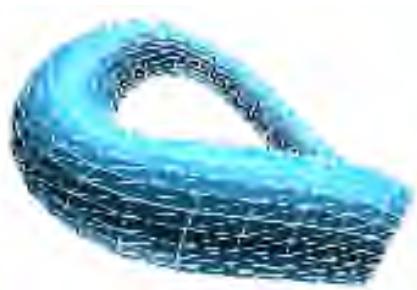
LatheBufferGeometry, LatheGeometry – класи для створення сіток з осьовою симетрією відносно осі Y (вази);



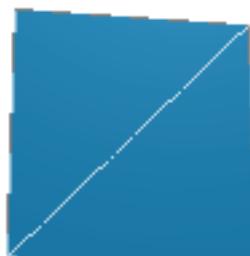
`OctahedronBufferGeometry`, `OctahedronGeometry` – класи правильних восьмигранників;



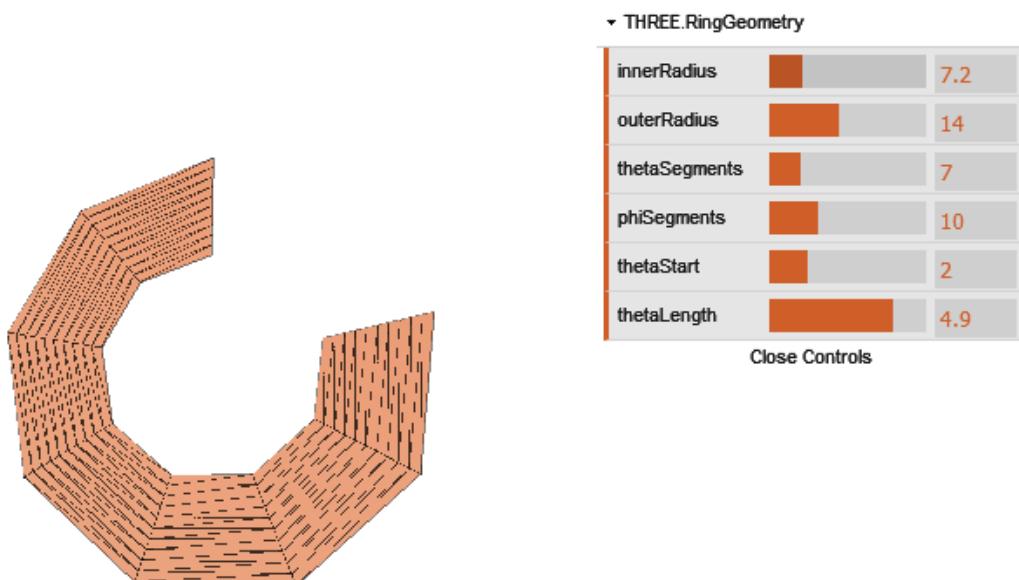
`ParametricBufferGeometry`, `ParametricGeometry` – класи параметричних поверхонь;



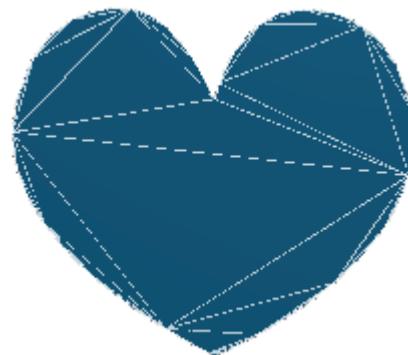
`PlaneBufferGeometry`, `PlaneGeometry` – класи площин;



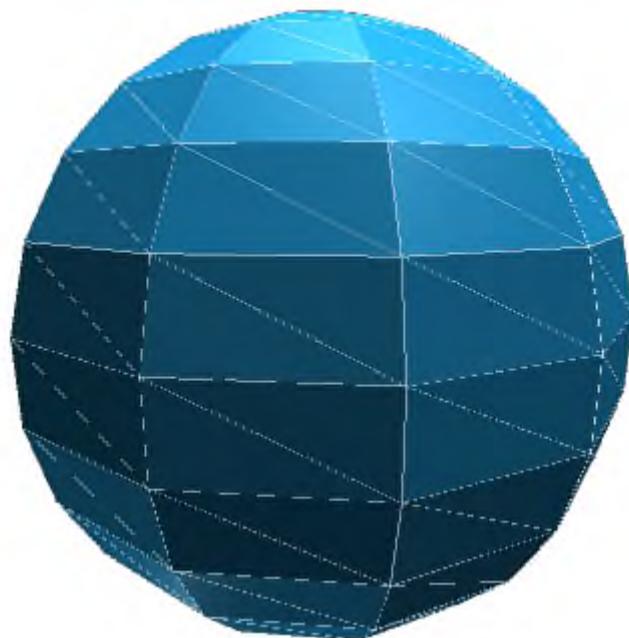
`RingBufferGeometry`, `RingGeometry` – класи для двовимірних кільцеподібних об'єктів;



`ShapeBufferGeometry`, `ShapeGeometry` – класи пласких об'єктів, побудованих за набором точок, з'єднаних лініями;

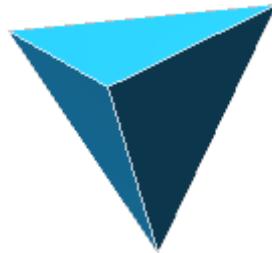


`SphereBufferGeometry`, `SphereGeometry` – класи сферичних об'єктів;



`TetrahedronBufferGeometry`, `TetrahedronGeometry` – класи

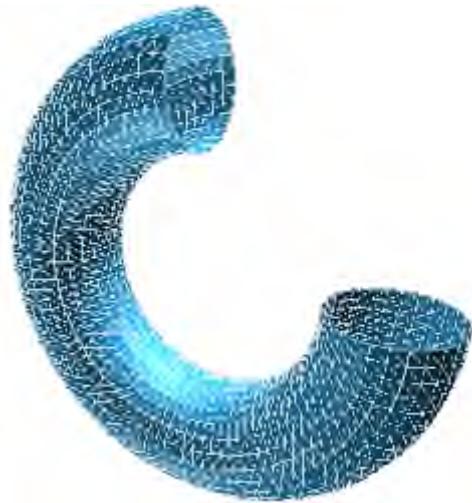
трикутних пірамід;



`TextBufferGeometry`, `TextGeometry` – класи текстових об'єктів;

## **TextBufferGeometry**

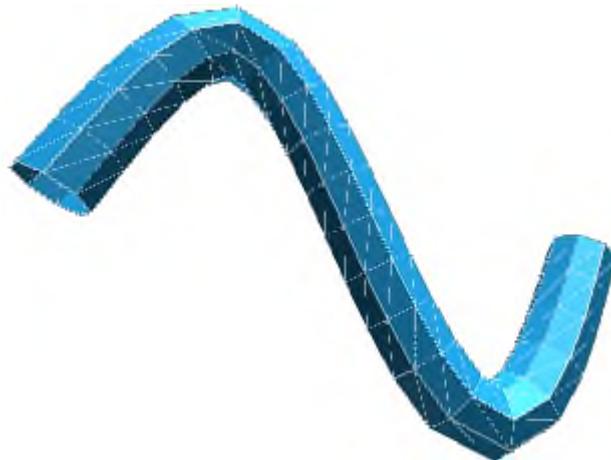
`TorusBufferGeometry`, `TorusGeometry` – класи торів;



`TorusKnotBufferGeometry`, `TorusKnotGeometry` – класи тороподібних фігур;

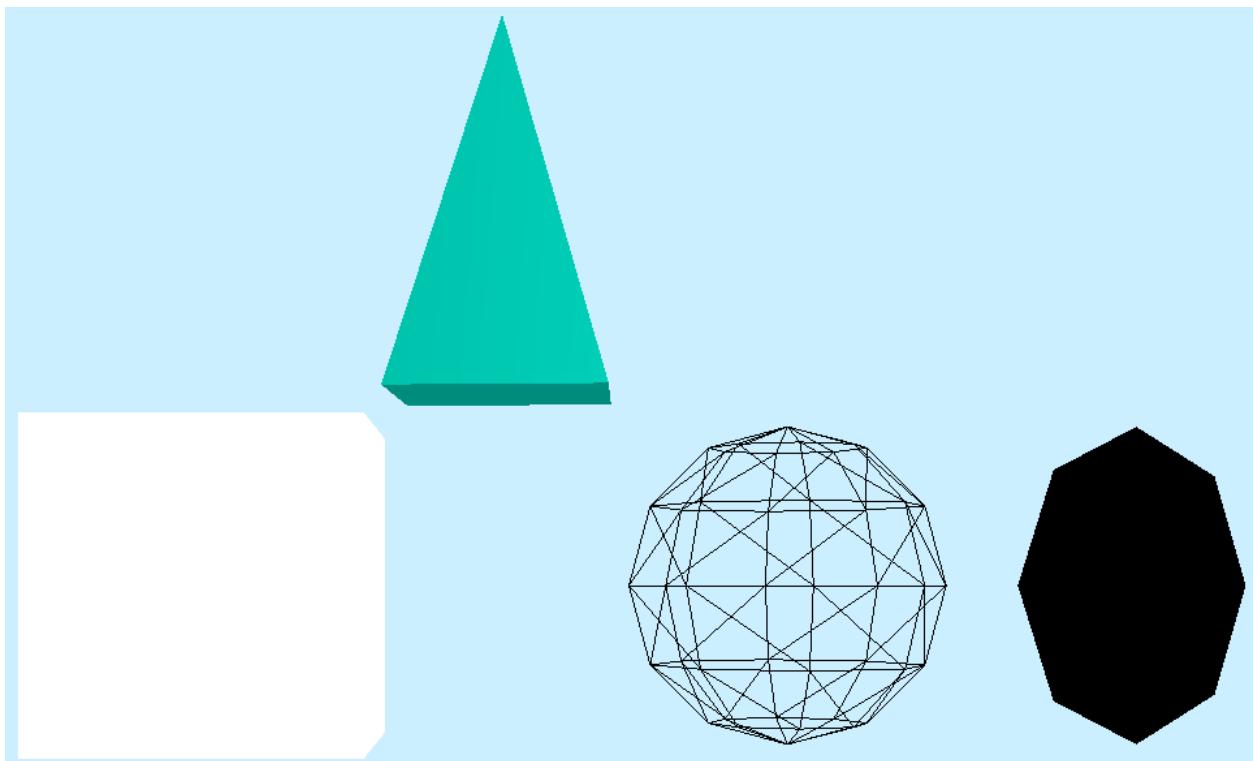


`TubeBufferGeometry`, `TubeGeometry` – класи трубок, витягнутих вздовж 3D-кривої.



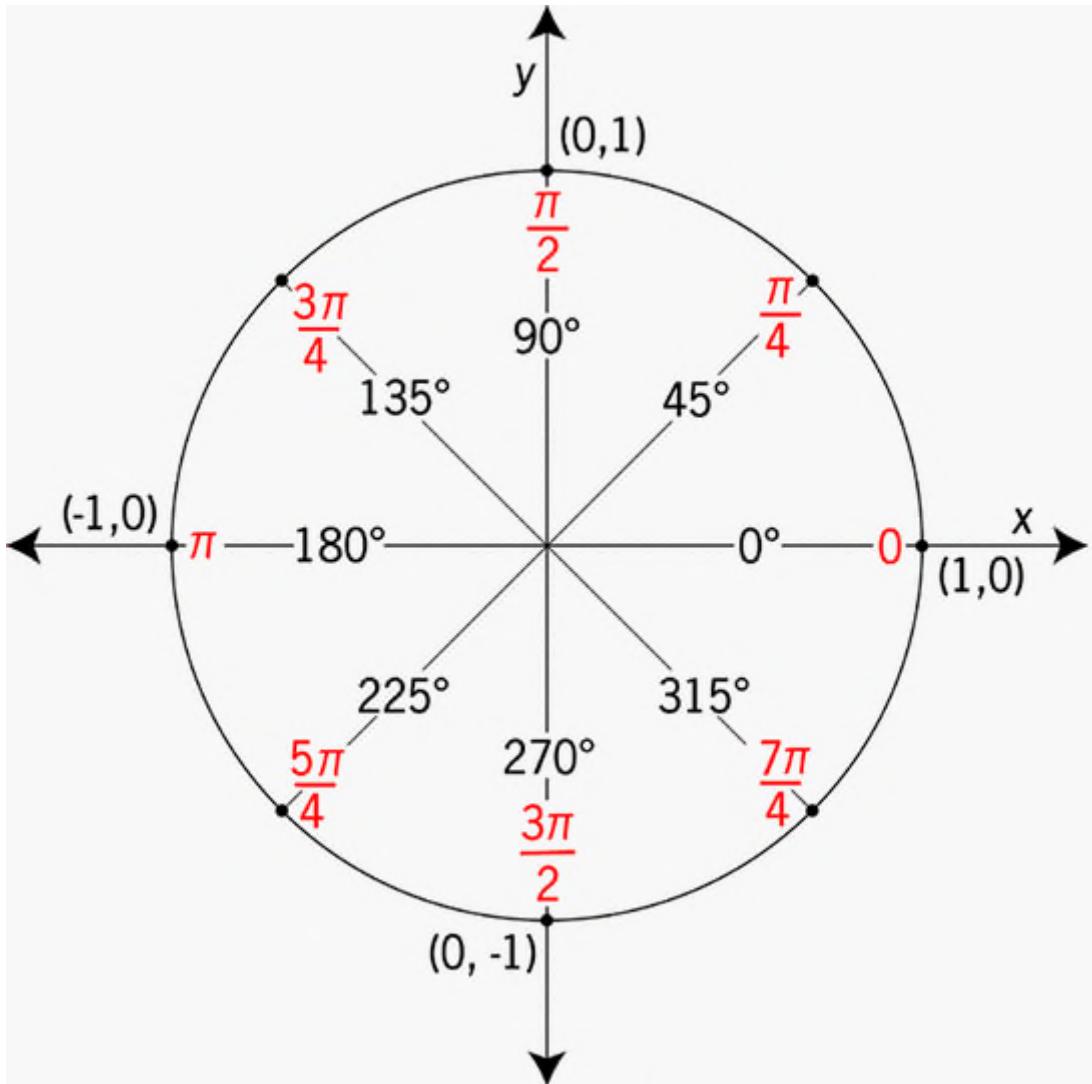
Продемонструємо створення плаского (2D) об'єкту:

```
var circlegeometry=new THREE.CircleBufferGeometry(0.5);
var circlematerial=new THREE.MeshBasicMaterial(
    {color: 0x000000});
var circlemesh=new THREE.Mesh(circlegeometry, circlematerial);
circlemesh.position.set(2, 0, -6);
circlemesh.rotation.set(0, 0.5, 0);
scene.add(circlemesh);
```



Обертання плаского об'єкту було виконане для того, щоб показати його «пласкість». Для визначення параметрів обертання доцільно скористатись

наступною схемою:



Для створення складних об'єктів доцільно використовувати їх параметричне подання, що вимагає визначення параметричної функції. У прикладі показано створення функції paraFunction, яка за двома параметрами  $a$  і  $b$  обчислює компоненти координатного вектору Vector3.

```
var paraFunction=function(a, b)
{
    var x=-5+5*a;
    var y=-5+5*b;
    var z=(Math.sin(a*Math.PI)+Math.sin(b*Math.PI))*(-7);

    return new THREE.Vector3(x, y, z);
}
```

```

var parageometry=new THREE.ParametricGeometry(paraFunction, 8, 8);
var paramaterial=new THREE.MeshBasicMaterial({color: 0xF3FFE2});
var paramesh=new THREE.Mesh(parageometry, paramaterial);
paramesh.position.set(0, -2, -100);
scene.add(paramesh);

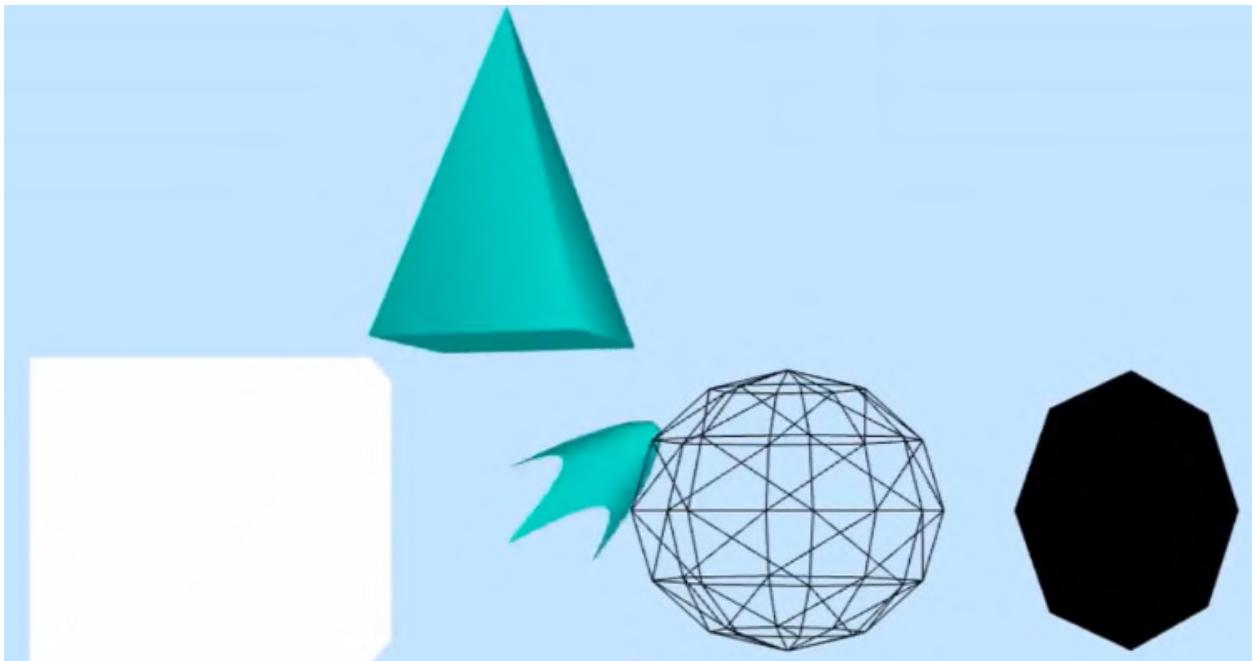
```

Для того, щоб анімувати цей об'єкт, додамо до функції render зміну його атрибутів:

```

{
    pyramidmesh.rotation.y+=0.1;
    paramesh.rotation.x+=0.1;
    paramesh.rotation.y+=0.1;
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}

```



Додамо до вже створених об'єктів площину:

```

var planegeometry=new THREE.PlaneGeometry(10, 10);
var planematerial=new THREE.MeshBasicMaterial();
var planemesh=new THREE.Mesh(planegeometry, planematerial);
planemesh.position.set(0, -20, -100);
scene.add(planemesh);

```

Для анімації площини скористаємся атрибутом vertices батьківського до PlaneGeometry класу – Geometry. Для цього додамо перед функцією render нову змінну

```
var delta=0;
```

а у тіло функції – код

```
delta+=0.1;
planegeometry.vertices[0].z=-25+Math.sin(delta)*50;
planegeometry.verticesNeedUpdate=true;
```

`vertices[0]` є об'єктом класу `Vector3`, що містить координату першої вершини прямокутника, який зображує площину. Для того, щоб виконана зміна `z`-координати вектору була відображенна, атрибут `verticesNeedUpdate` необхідно встановити у значення `true`.

Аналогічно можна змінити такі параметри геометрії об'єкту, як масив кольорів `colors` (після чого встановити `colorsNeedUpdate` у `true`), полігонів `faces` (`elementsNeedUpdate`), текстур `faceVertexUvs` (`uvsNeedUpdate`) та ін.

### 3.5. Створення матеріалів

`Three.js` надає можливість використання таких основних типів матеріалів:

`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:

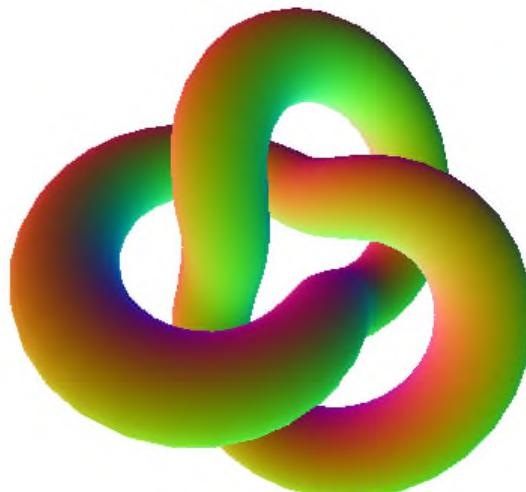
ближчі до камери частини об'єкту світліші, дальші – темніші.



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



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



`MeshPhongMaterial` – матеріал для блискучих 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);

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

`SpriteMaterial` – матеріал для об'єктів класу `Sprite`;

`MeshPhysicalMaterial` – розширення `MeshStandardMaterial`, що

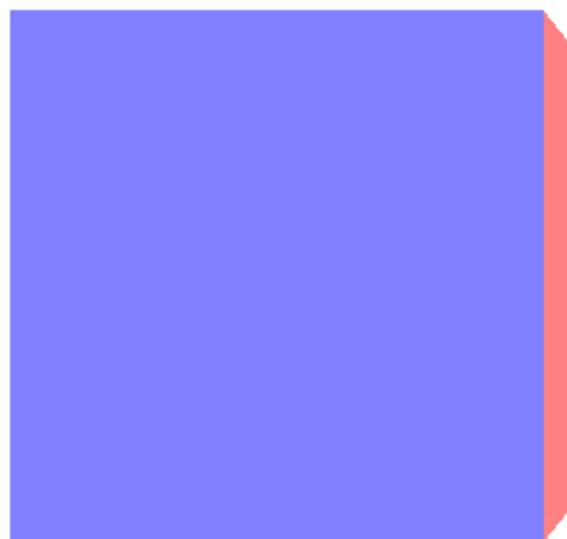
надає більший контроль за відбивною здатністю.



Поекспериментуємо із деякими матеріалів. Розпочнемо із заміни матеріалу для прямокутного паралелепіпеда на MeshNormalMaterial червоного кольору, встановивши атрибути прозорості `transparent` у `true` та непрозорості `opacity` у `1`:

```
var boxmaterial=new THREE.MeshNormalMaterial({
    color: 0xFF0000,
    transparent: true,
    opacity: 1
});
```

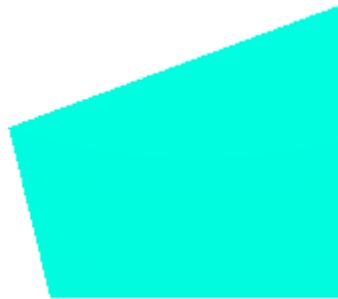
Для даного типу матеріалу колір залежить від того, з якого боку на нього падає світло.



Для площини використаємо MeshPhongMaterial, що реалізує затемнення

за Фонгом:

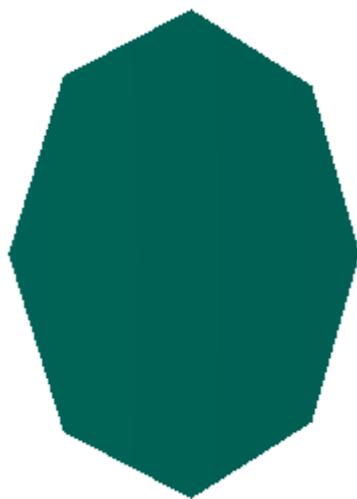
```
var planematerial=new THREE.MeshPhongMaterial({
    color: 0xF3FFE2,
    specular: 0xFF0000,
    shininess: 50
});
```



Для плаского кола оберемо MeshStandardMaterial, надавши йому зеленого кольору:

```
var circlematerial=new THREE.MeshStandardMaterial({
    color: 0x098877,
    roughness: 90.0,
    metalness: 0.2
});
```

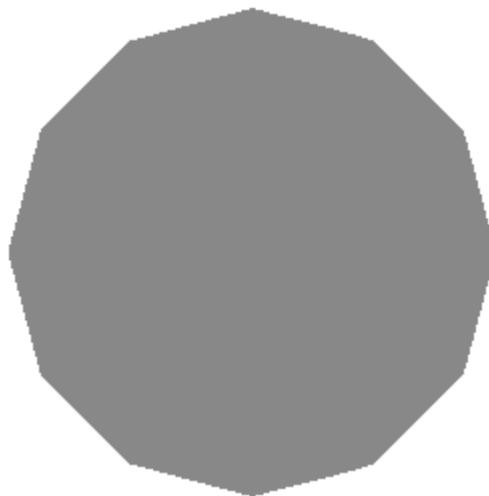
Більші значення параметру металевості (metalness) призводять до затемнення об'єкту, вища шорсткість (roughness) – до зміни кольору.



Останнім змінимо матеріал сфери:

```
var spherematerial=new THREE.LineBasicMaterial({
    color: 0x888888
```

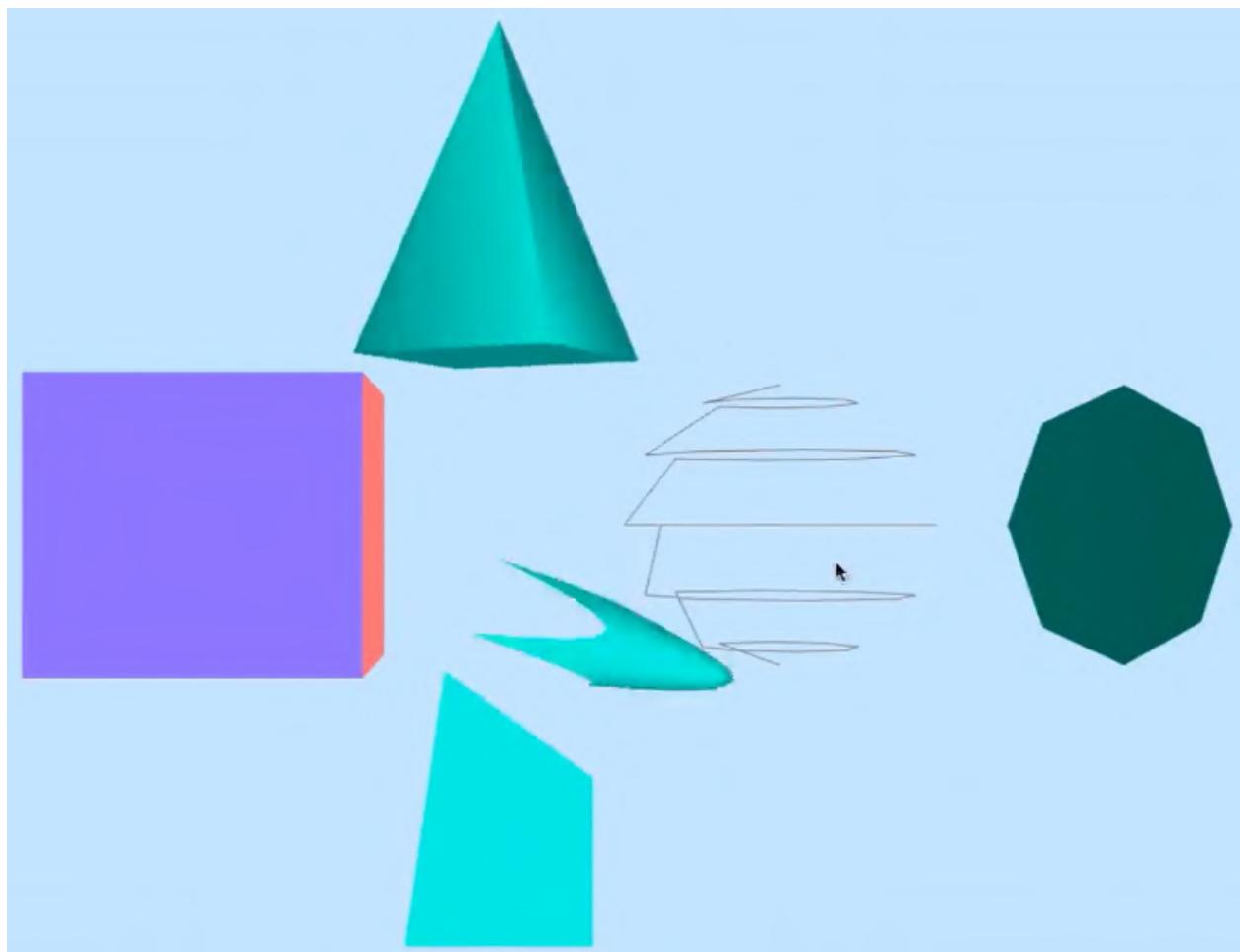
```
});
```



Колір сфери змінюється, проте сутність даного матеріалу втрачена через те, що об'єднання геометрії та матеріалу було виконано за допомогою сіткової моделі (mesh). Замінимо її на лінійну (Line):

```
var spheremesh=new THREE.Line(spheregeometry, spherematerial);
```

Після цього лінія стає основним об'єктом, що утворює сферу.



### 3.6. Створення шейдерів

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

Ураховуючи, що попередня сцена дещо перевантажена об'єктами, для зручності експериментування створимо новий файл shaders.html:

```
<!DOCTYPE html>
<html>
    <head><title>threeJS shaders</title></head>
    <body>
        <canvas id="Canvas"></canvas>
        <script src="three.js-dev/build/three.js"></script>
    <!-- Вершинний шейдер -->
```

```
<!-- Фрагментний шейдер -->

<script>
    var renderer=new THREE.WebGLRenderer({canvas:
document.getElementById('Canvas'), antialias: true});
    renderer.setClearColor(0xCBEFFF);
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(window.innerWidth, window.innerHeight);

    var camera=new THREE.PerspectiveCamera(35,
        window.innerWidth / window.innerHeight, 0.1, 3000);

    var scene=new THREE.Scene();

    var lightOne=new THREE.AmbientLight(0xfffff, 0.5);
    scene.add(lightOne);

    var lightTwo=new THREE.PointLight(0xfffff, 0.5);
    scene.add(lightTwo);

    // користувацький шейдер
    var boxgeometry=new THREE.BoxBufferGeometry(100, 100, 100,
                                                10, 10, 10);
    var boxmesh=new THREE.Mesh(boxgeometry, customMat);
    boxmesh.position.set(-100, 0, -1000);
    scene.add(boxmesh);

    var cubegeometry=new THREE.CubeGeometry(100, 100, 100);
    var cubemesh=new THREE.Mesh(cubegeometry, customMat);
    cubemesh.position.set(150, 0, -1000);
    scene.add(cubemesh);

    requestAnimationFrame(render);

    function render(){
        renderer.render(scene, camera);
```

```

        requestAnimationFrame(render);
    }
</script>
</body>
</html>
```

Обидва об'єкти, розміщені на сцені – кубічні, проте створені із використанням різних геометрій. Для того, щоб сцена стала робочою, на ній відведено місце для вершинного та фрагментного шейдерів, за допомогою яких створюється користувацький `customMat`.

Спочатку створимо вершинний шейдер. Для цього додамо його опис шейдерною мовою GLSL (OpenGL Shading Language), встановивши для цього значення атрибуту `type` у `x-shader/x-vertex`:

```

<!-- Вершинний шейдер -->
<script type="x-shader/x-vertex" id="vertexShader">
    void main()
    {
        gl_Position = projectionMatrix *
                      modelViewMatrix * vec4(position,1.0);
    }
</script>
```

Вершинний шейдер по черзі отримує кожну із вершин і опрацьовує їх. 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.

Фрагментний шейдер визначатиме колір встановленням `gl_FragColor` у чотиривимірний вектор, координати якого представлятимуть червоний, зелений, жовтий і синій кольори відповідно:

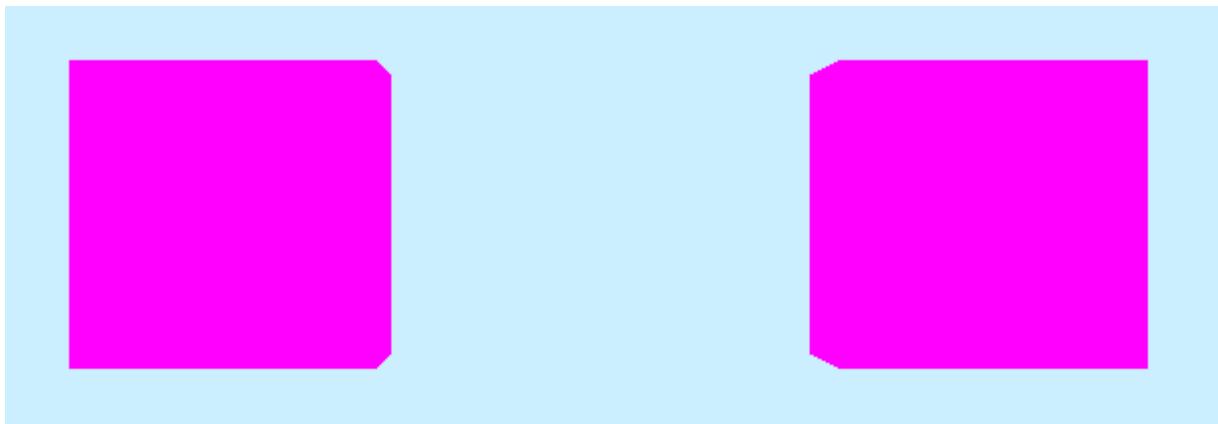
```

<script type="x-shader/x-fragment" id="fragmentShader">
    void main()
    {
        gl_FragColor = vec4(5.0, 0.0, 5.0, 0.0);
    }
</script>
```

Для створення користувацького шейдеру визначимо змінну `customMat` в основний програмі як об'єкт класу `ShaderMaterial`:

```
// користувацький шейдер
var customMat=new THREE.ShaderMaterial({
    vertexShader: document.getElementById('vertexShader').textContent,
    fragmentShader: document.getElementById('fragmentShader').textContent
});
```

`ShaderMaterial` надав створити спеціальний матеріал `customMat` – користувацький шейдер – за допомогою вершинного та фрагментного шейдерів.



Продемонструємо можливість інтерфейсу між програмами мовами GLSL та JavaScript. Для цього додамо до фрагментного шейдеру змінну `delta` та визначимо їй у відповідність одну з масштабних координат:

```
<script type="x-shader/x-fragment" id="fragmentShader">
    uniform float delta;
    void main()
    {
        gl_FragColor = vec4(5.0, delta + 0.0, 5.0, 0.0);
    }
</script>
```

У описі користувацького шейдера мовою JavaScript визначимо її початкове значення:

```
var customUniforms={
    delta: {value: 0}
};
var customMat=new THREE.ShaderMaterial({
    uniforms: customUniforms,
    vertexShader: document.getElementById('vertexShader').textContent,
    fragmentShader: document.getElementById('fragmentShader').textContent
});
```

Для того, щоб змінити шейдерну змінну delta, слід звернутися до відповідної властивості `boxmesh.material.uniforms.delta.value`:

```
var delta=0;
function render(){
    delta+=0.1;
    boxmesh.material.uniforms.delta.value=0.5+Math.sin(delta);
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}
```

### 3.7. Джерела світла та камери

За допомогою Three.js створимо у файлі `LightCamera.html` сцену із площиною, на якій розташовано конічний та кубічний об'єкти:

```
<!DOCTYPE html>
<html>
    <head><title>LightCamera</title></head>
    <body>
        <canvas id="Canvas"></canvas>
        <script src="three.js-dev/build/three.js"></script>
<script>
    var renderer = new THREE.WebGLRenderer(
        {canvas: document.getElementById('Canvas'), antialias: true});
    renderer.setClearColor(0x333333);
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(window.innerWidth, window.innerHeight);

    //перспективна камера
    var camera=new THREE.PerspectiveCamera(35,
        window.innerWidth / window.innerHeight, 0.1, 20000);
    camera.position.z = 500;

    //ортографічна камера

    var scene = new THREE.Scene();

    var conegeometry = new THREE.CylinderGeometry(100,200,100,500);
```

```
var conematerial = new THREE.MeshLambertMaterial();
var conemesh = new THREE.Mesh(conegeometry, conematerial);
conemesh.position.set(-150, -50, -500);
conemesh.rotation.x = -100;
scene.add(conemesh);

var cubegeometry = new THREE.CubeGeometry(90, 100, 90, 200);
var cubematerial = new THREE.MeshLambertMaterial();
var cubemesh = new THREE.Mesh(cubegeometry, cubematerial);
cubemesh.position.set(200, -50, -500);
cubemesh.rotation.x = -100;
scene.add(cubemesh);

var planegeometry = new THREE.PlaneGeometry(900, 500, 300, 900);
var planematerial = new THREE.MeshPhongMaterial(
    {color: 0x333333});
var planemesh = new THREE.Mesh(planegeometry, planematerial);
planemesh.position.set(50, -70, -500);
planemesh.rotation.set(-290, 0, 0);
scene.add(planemesh);

//рівномірне освітлення
var lightOne = new THREE.AmbientLight(0xffffff, 0.5);
lightOne.position.z = 300;
scene.add(lightOne);

//точкове освітлення

//напрямлене освітлення

//прожекторне освітлення

//напівсферичне освітлення

//відкидання тіней
```

```
requestAnimationFrame(render);

function render() {
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}

</script>
</body>
</html>
```



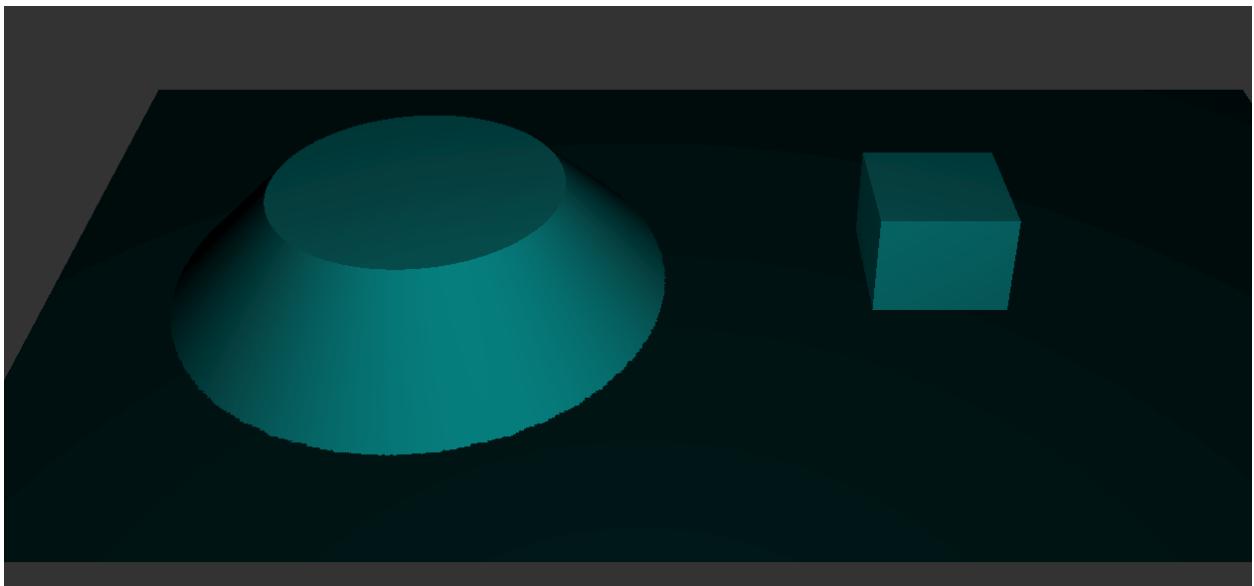
На сцені є заготовки для різних джерел світла (активоване рівномірне освітлення) та камер (включена перспективна камера).

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:



Закоментуємо фрагмент коду, що відноситься до розсіяного навколошнього освітлення та створимо об'єкт класу `PointLight`:

```
// точкове освітлення
var lightOne = new THREE.PointLight(0xfffff, 0.5);
scene.add(lightOne);
```



Конструктор класу `PointLight` має чотири параметри: колір (за замовчанням білий – `0xfffff`), інтенсивність (за замовчанням максимальну – 1), дальність світла (за замовчанням 0 – без обмежень), коефіцієнт затухання світла зі збільшенням відстані (за замовчанням 1).

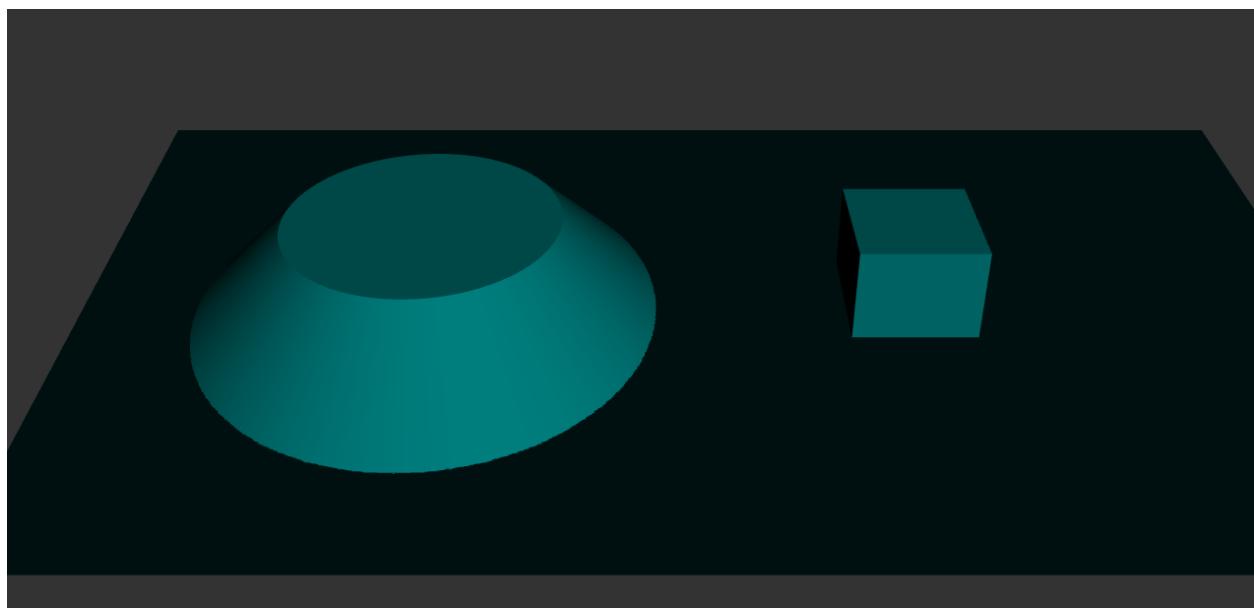
Точкове освітлення залежить від того, де знаходиться джерело освітлення. Продемонструємо це, циклічно змінюючи координати джерела

освітлення так, щоб воно оберталось по колу із центром в початку координат радіусом 300 в площині YZ:

```
var t=0;
function render() {
    lightOne.position.y = 300*Math.sin(t);
    lightOne.position.z = 300*Math.cos(t);
    t += 0.01;
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}
```

Напрямлене освітлення – це світло, яке випромінюється у певному напрямі. Це джерело світла буде вести себе так, ніби воно знаходиться нескінченно далеко, і промені йдуть паралельно. Для такого освітлення встановлення атрибуту `rotation` не має сенсу – напрям освітлення визначається об'єктом (`target`), що буде освітлюватись. Так, прибравши попереднє джерело світла, можемо включити нове:

```
//напрямлене освітлення
var lightOne = new THREE.DirectionalLight(0xfffff, 0.5, 1000);
lightOne.target = conemesh;
scene.add(lightOne);
```



На сцені видно, що освітлення конічного об'єкту відрізняється від освітлення кубічного. Створимо ефект зміни освітлюваного об'єкту за допомогою опрацювання натискання кнопок миші:

```

function changeTarget(){
    if(lightOne.target == conemesh)
        lightOne.target = cubemesh;
    else
        lightOne.target = conemesh;
    requestAnimationFrame(render);
}

document.getElementById("Canvas").onclick = changeTarget;

```

Функція `changeTarget` перевіряє, на який об'єкт спрямоване освітлення, та встановлює інший. Для того, щоб зміна освітлення відобразилась на сцені, ініціюємо рендеринг сцени викликом `requestAnimationFrame(render)`.

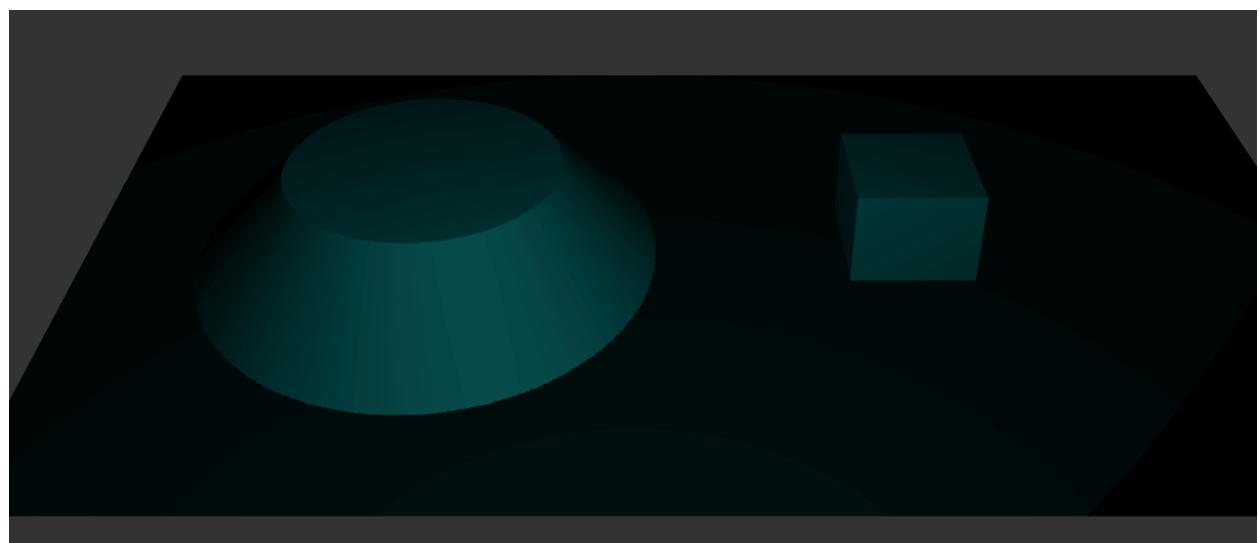
Виклик `changeTarget` відбувається завдяки встановленню значення властивості полотна `onclick`, що відповідає за опрацювання натискань на кнопки миші. Таку саму властивість мають й інші об'єкти сцени, що надає можливість зручного керування ними.

Так само, як і раніше, закоментуємо останній код та створимо нове джерело світла як об'єкт класу `SpotLight`:

```

//прожекторне освітлення
var lightOne = new THREE.SpotLight(0xfffff, 0.5, 1000);
lightOne.target = conemesh;
scene.add(lightOne);

```

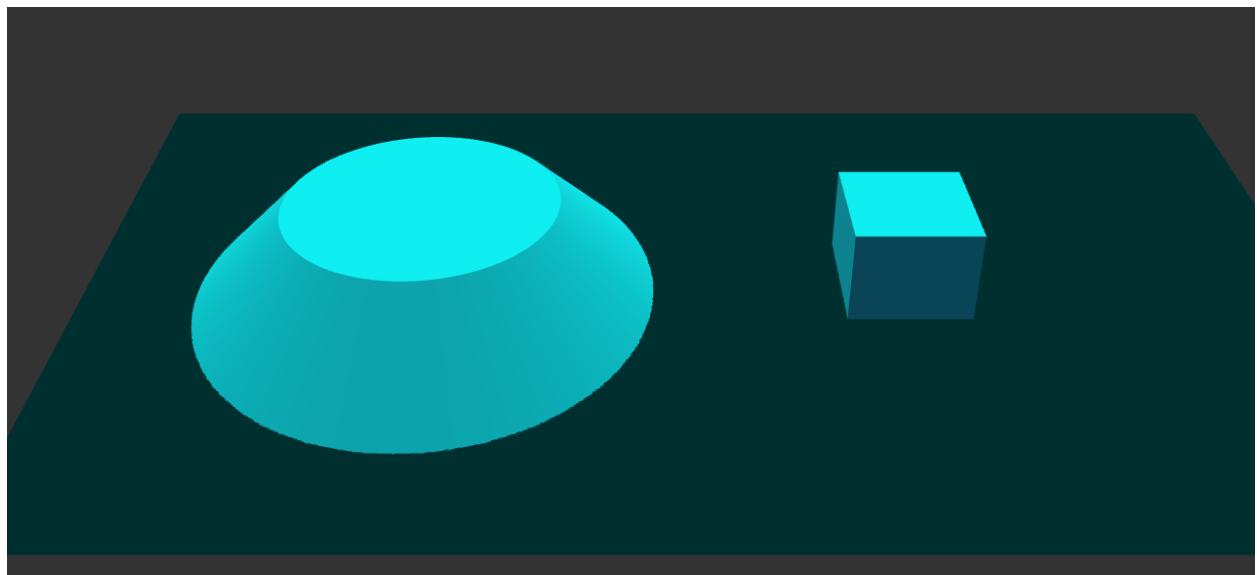


Так само, як і для напрямленого освітлення, прожекторне потребує вказання освітлюваного об'єкту (конічного у наведеному фрагменті).

Конструктор об'єкту напівсферичного освітлення приймає три параметри – колір неба, колір землі та інтенсивність освітлення:

```
//напівсферичне освітлення
var lightOne = new THREE.HemisphereLight(0xfffffff, 0x080820, 1);
scene.add(lightOne);
```

Напівсферичне освітлення знаходиться над сценою, а колір освітлення змінюється від кольору неба до кольору землі:



Для того, щоб створити тінь, необхідно звернутись до візуалізатора (рендерера), активувати тіньову карту та встановити її тип:

```
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.PCFShadowMap;
```

Властивість `shadowMap.type` визначає тип тіньової карти. Можливі значення: `THREE.BasicShadowMap` (без фільтру – швидкі та низькоякісні), `THREE.PCFShadowMap` (з фільтром відсоткового наближення – Percentage-Closer Filtering) та `THREE.PCFSOFTShadowMap` (з фільтром відсоткового наближення та білінійною фільтрацією в шейдері – Percentage-Closer Soft Shadows).

Далі визначимо точкове джерело світла, його позицію та спрямуємо на конічний об'єкт:

```
var lightOne = new THREE.SpotLight(0xfffffff, 2.0, 3000);
lightOne.position.x = 50;
lightOne.target = conemesh;
```

Для відкидання динамічних тіней встановимо властивість `castShadow` об'єкту `lightOne` у `true`:

```
lightOne.castShadow = true;
```

Клас `LightShadow` застосовується для розрахунку тіней усередині `PointLight`, а також є базовим класом для інших класів тіней.

```
lightOne.shadow = new THREE.LightShadow(
    new THREE.PerspectiveCamera(100, 1, 500, 1000));
```

Властивість `bias` об'єкту `lightOne.shadow` визначає відхилення від тіньової карти (скільки можна додати або відняти від нормалізованої глибини при визначенні, чи знаходиться поверхня у тіні). Значення за замовчуванням дорівнює 0. Найменші підстроювання значення даного властивості (блізько 0.0001) можуть допомогти зменшити артефакти в тінях:

```
lightOne.shadow.bias = 0.0001;
```

Властивість `mapSize` є об'єктом класу `Vector2`, який визначає ширину і висоту тіньової карти. Більш високі значення дають якісніші тіні за рахунок збільшення часу обчислення. Значення повинні бути ступенями двійки, аж до значення властивості `WebGLRenderer.capabilities.maxTextureSize` даного пристрою, причому ширина і висота можуть бути неоднаковими (так наприклад, (512, 1024) цілком допустимі). Значення за замовчанням – (512, 512):

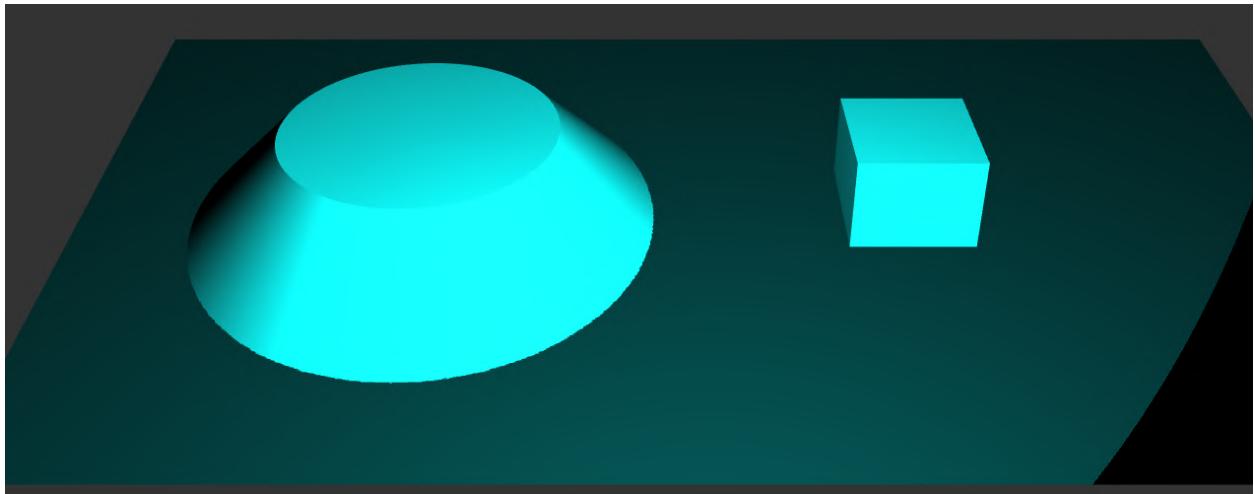
```
lightOne.shadow.mapSize.width = 2048 * 2;
lightOne.shadow.mapSize.height = 2048 * 2;
scene.add(lightOne);
```

Після цього на сцені з'являються різні тіньові ефекти, у тому числі на площині (з правого боку), конусі (заду). На куб спрямоване джерело світла, тому він не має жодного затінення (за винятком лівої грані – це пов'язано із затіненням конуса, що знаходиться поряд).

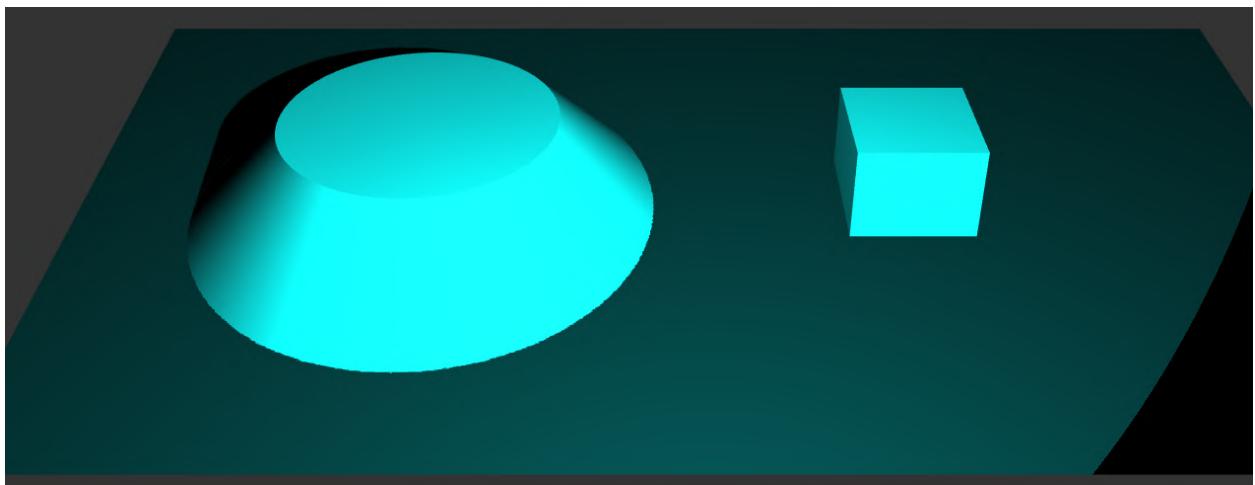
Якщо властивість `castShadow` відповідає за відкидання тіней, то `receiveShadow` – за їх прийняття:

```
conemesh.castShadow = true;
cubemesh.castShadow = true;
```

```
planemesh.receiveShadow = true;
```



Наведений фрагмент надає можливість відображати тіні від конічного та кубічного об'єктів на площині.



Так само, як й у попередніх випадках, закоментуємо пов'язаний із тіньовими ефектами код, включимо раніше вимкнуте точкове джерело освітлення та перейдемо до експериментів із камерою:

```
//точкове освітлення
var lightOne=new THREE.PointLight(0xfffff, 0.5);
scene.add(lightOne);
```

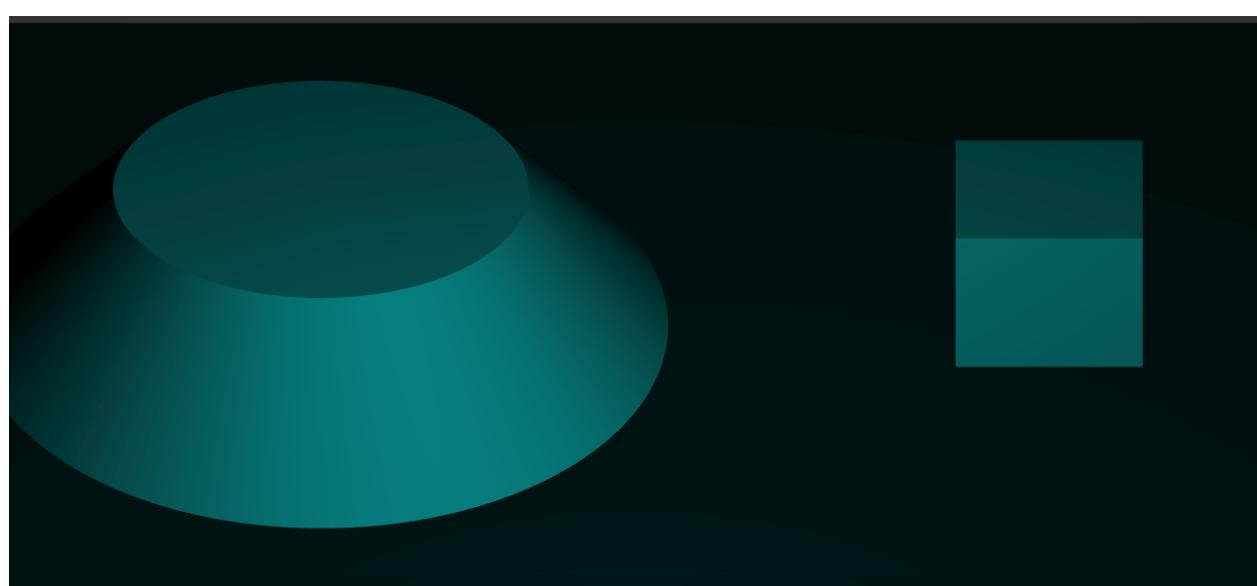
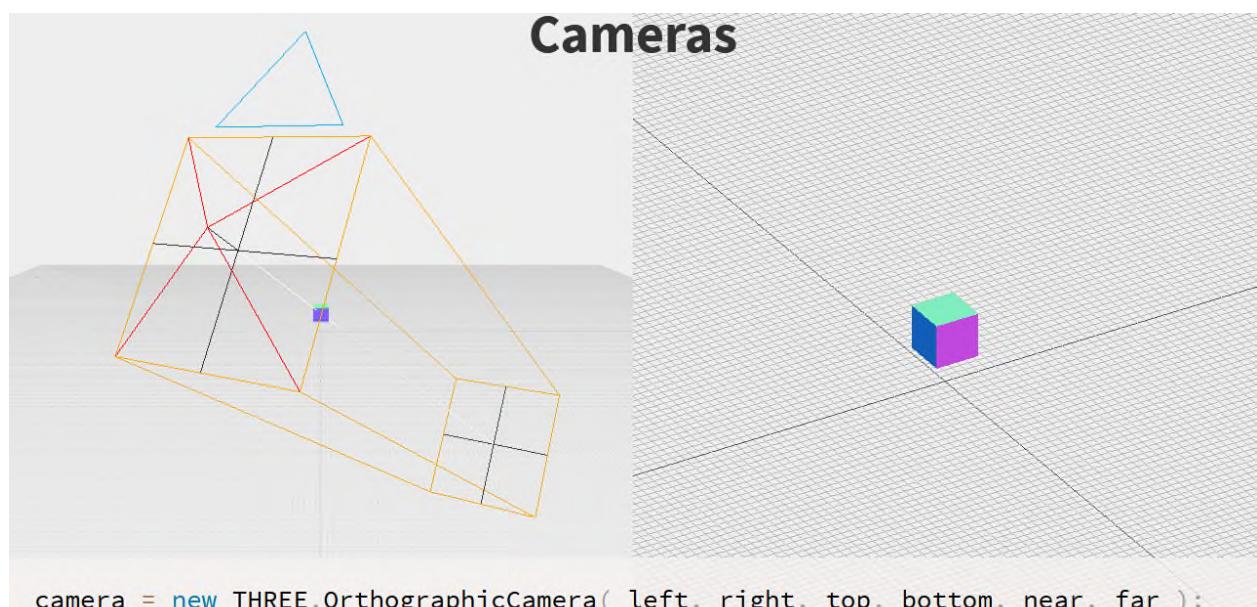
Камера, що використовується на сцені – камера з перспективною проекцією PerspectiveCamera. Даний режим проекції призначений, щоб найбільш повно зімітувати людський зір. Це найпоширеніший режим проектування, використовуваний для візуалізації (рендеринга) тривимірної сцени.

Замінимо перспективну камеру на ортографічну:

```
//перспективна камера
//var camera=new THREE.PerspectiveCamera(35, window.innerWidth / window.innerHeight, 0.1, 2000);
//camera.position.z = 500;

//ортографічна камера
var camera = new THREE.OrthographicCamera(-300, 300, 200, -200, 0.1, 2000);
```

Ортографічна камера не має перспективи і має асиметричний вигляд: за ортографічного проектування розмір об'єкта у видимому зображенні залишається постійним, незалежно від відстані між ним і камерою. Це може бути корисним при відображення двомірних сцен і елементів інтерфейсу користувача.



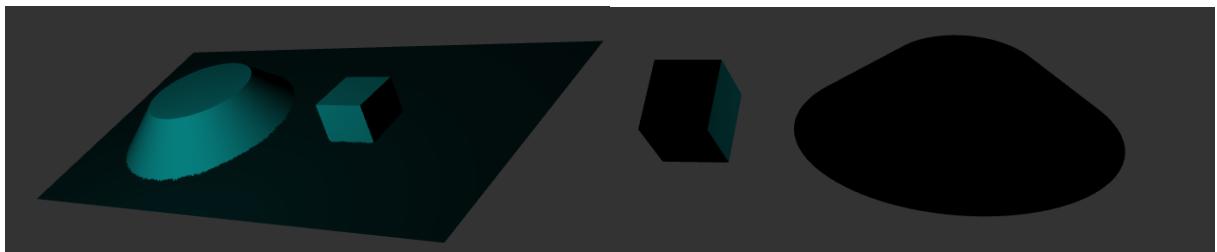
Конструктор класу OrthographicCamera(left, right, top, bottom, near, far) приймає 6 параметрів відсікання області видимості камери: left

– ліва площа, `right` – права площа, `top` – верхня площа, `bottom` – нижня площа, `near` – близня площа, `far` – дальня площа відсікання області видимості камери. Разом вони визначають область перегляду камери (у вигляді усіченого піраміди):

Повернемося до використання звичайної камери, спрямуємо її на джерело освітлення та ініціюємо обертання навколо вісі Y:

```
var t=0;
function render() {
    //lightOne.position.y = 300*Math.sin(t);
    //lightOne.position.z = 300*Math.cos(t);
    t += 0.01;
    camera.lookAt(lightOne.position);
    camera.position.x = Math.sin(t) * 2000;
    camera.position.z = Math.cos(t) * 2000;
    renderer.render(scene, camera);
    requestAnimationFrame(render);
}
```

У результаті отримаємо ілюзію обертання об'єктів сцени, яке досягається завдяки руху камери.



Якщо скористатись ортографічною камерою, складається враження, що об'єкти проходять перед камерою, а через деякий час повертаються – насправді ж камера знову обертається навколо самих об'єктів.

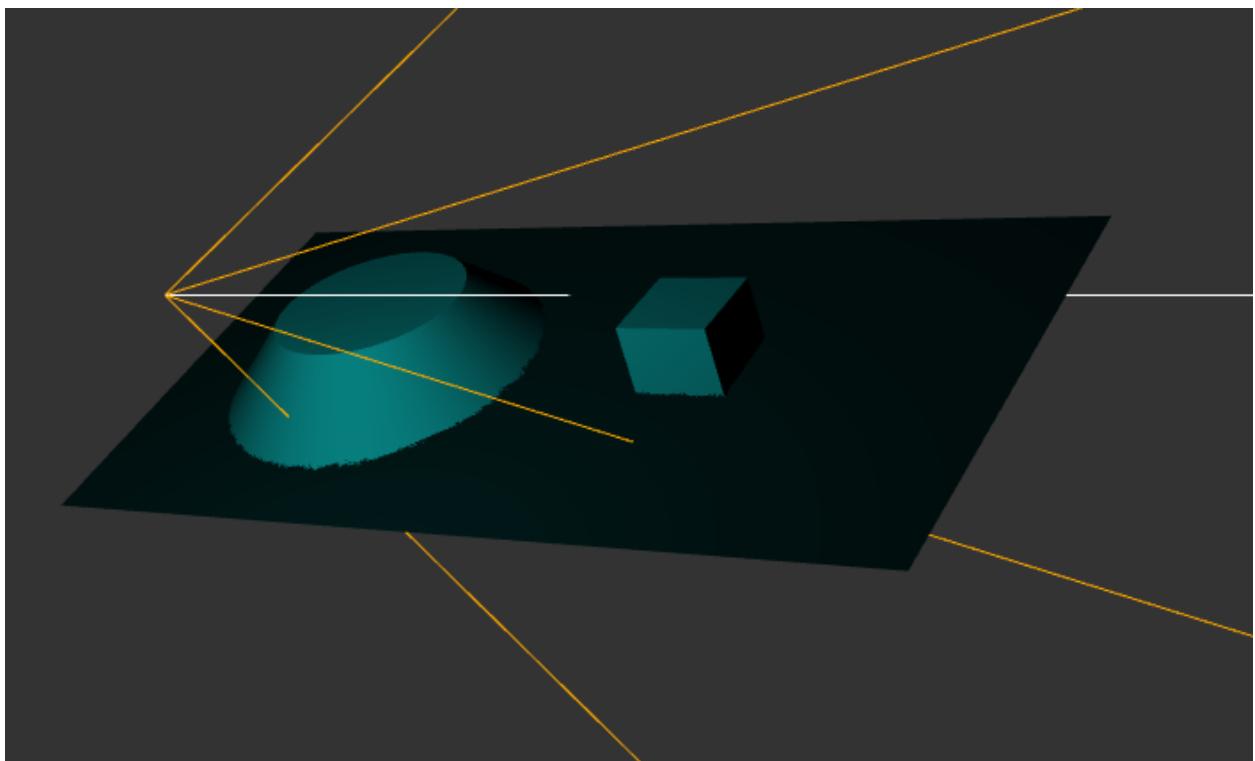
Розглянемо деякі допоміжні методи. Для демонстрації допомоги при роботі з камерою додамо до сцени другу камеру, розташувавши її всередині сцени:

```
var secondCamera = new THREE.PerspectiveCamera(35,
    window.innerWidth / window.innerHeight, 0.1, 20000);
var cameraHelper = new THREE.CameraHelper(secondCamera);
scene.add(cameraHelper);
```

`CameraHelper` – допоміжний клас для визначення того, що потрапляє в

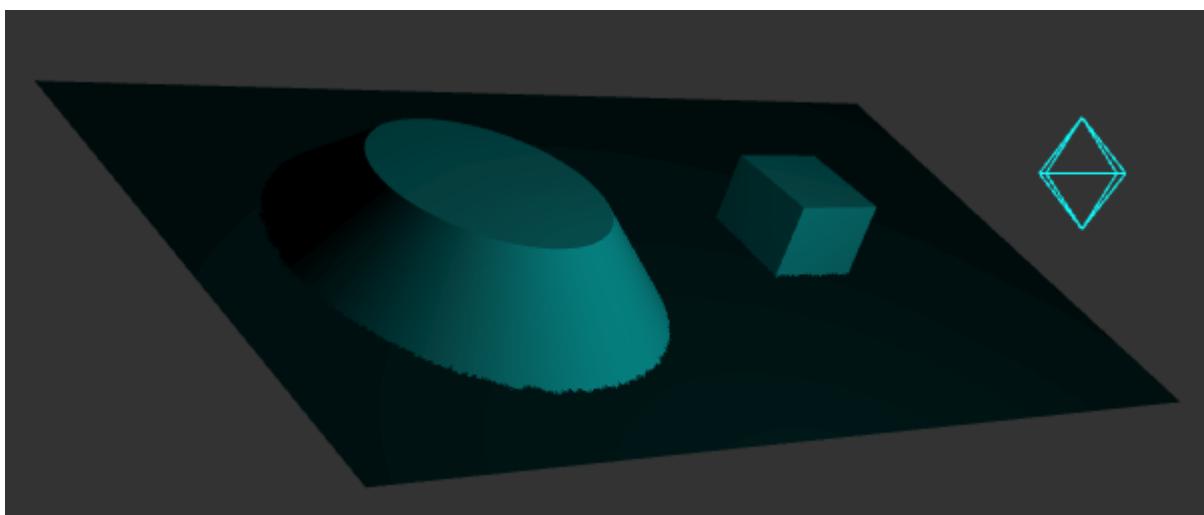
обсяг усіченої піраміди, який буде бачити камера. Область видимості камери показується за допомогою LineSegments.

У результаті отримаємо гарну можливість для налагодження – можна легко перевірити, де знаходиться камера.



Для налагодження світла застосуємо клас PointLightHelper:

```
var lightHelper=new THREE.PointLightHelper(lightOne, 40);
scene.add(lightHelper);
```



Клас PointLightHelper створює видимий допоміжний елемент, що складається зі сферичної сітки для відображення зони освітлення від PointLight – точкового освітлення. Конструктор PointLightHelper приймає

три параметри: джерело світла, яке потрібно візуалізувати; розмір сфери допоміжного елементу (значення за замовчуванням дорівнює 1) та колір освітлення (додатковий, необов'язковий параметр).

Наступний допоміжний об'єкт буде для спрямованого світла, яке необхідно активувати замість точкового:

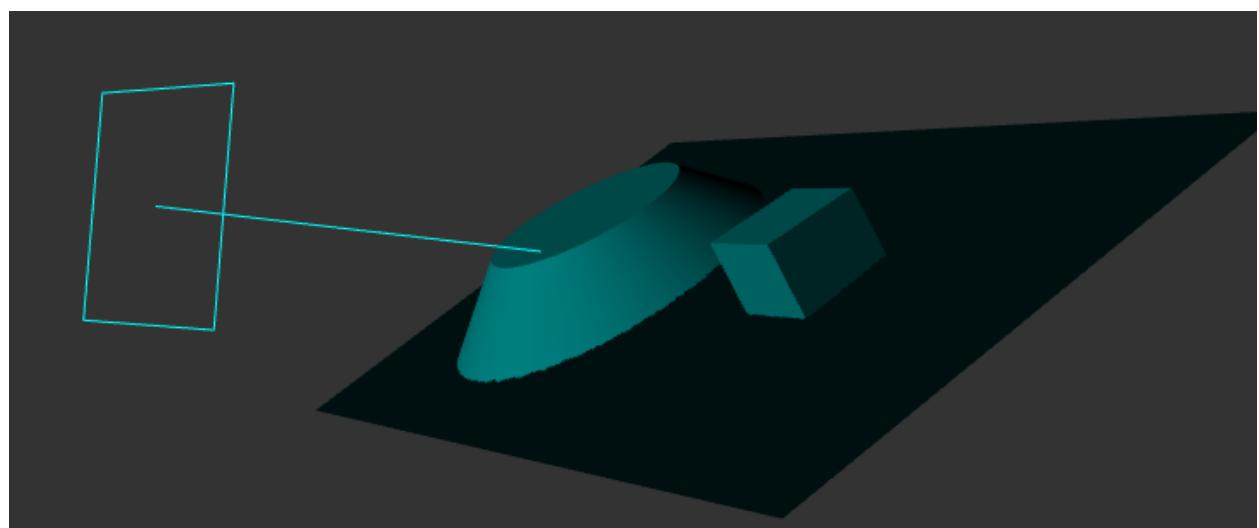
```
//напрямлене освітлення
var lightOne = new THREE.DirectionalLight(0xffff, 0.5, 1000);
lightOne.target = conemesh;
scene.add(lightOne);

var lightHelper=new THREE.DirectionalLightHelper(lightOne, 100);
scene.add(lightHelper);
```

Клас `DirectionalLightHelper` створює допоміжний об'єкт, що складається із площини та лінії, які представляють положення і напрямок світла. Конструктор має три параметри: джерело світла, яке візуалізується, розмір площини (необов'язковий параметр – значення за замовчуванням дорівнює 1) та колір.

Для того, щоб даний допоміжний об'єкт правильно анімувався, до функції `render` необхідно додати виклик методу оновлення допоміжного елемента відповідно до позиції і напряму відображуваного спрямованого світла:

```
lightHelper.update();
```



Наступний крок – налаштування керованого користувачем 360-градусного огляду сцени всередині браузера. Для цього скористаємось

прикладом OrbitControls.js, що поставляється разом із бібліотекою Three.js. У каталогі, в якому знаходиться файл CamerLight.html, створимо новий файл OrbitControls.html та каталог js, до якого скопіюємо файли three.js-dev/examples/js/controls/OrbitControls.js та three.js-dev/build/three.min.js.

360-градусний панорамного огляд, що регулюється за допомогою миši, вимагає визначення двох спеціальних функцій – init() та animate():

```
<!DOCTYPE html>
<html>
    <head><title>OrbitControls</title></head>
    <body>
        <script src="js/three.min.js"></script>
        <script src="js/OrbitControls.js"></script>
<script>
    init();
    animate();

</script>
</body>
</html>
```

У html-файлі цього разу не визначено полотно – воно буде створено динамічно у функції init(), зміст якої є достатньо традиційним, адже значна її частина містить код із попереднього прикладу:

```
function init() {
    scene = new THREE.Scene();
    var WIDTH = window.innerWidth, HEIGHT = window.innerHeight;

    renderer = new THREE.WebGLRenderer({antialias: true});
    renderer.setSize(WIDTH, HEIGHT);
    document.body.appendChild(renderer.domElement);

    camera = new THREE.PerspectiveCamera(45, WIDTH/ HEIGHT, 0.1,
20000);
    camera.position.set(0, 6, 0);
    scene.add(camera);

    window.addEventListener('resize', function() {
```

```

var WIDTH = window.innerWidth, HEIGHT = window.innerHeight;
renderer.setSize(WIDTH, HEIGHT);
camera.aspect = WIDTH / HEIGHT;
camera.updateProjectMatrix();
} );

renderer.setClearColor(0x333F47, 1);

var light = new THREE.PointLight(0xffffffff);
light.position.set(-100, 200, 100);
scene.add(light);

var cylgeometry = new THREE.CylinderGeometry(3, 3, 7, 7);
var cylmaterial = new THREE.MeshLambertMaterial();
var cylmesh = new THREE.Mesh(cylgeometry, cylmaterial);
cylmesh.position.set(0.9, -5, -6);
scene.add(cylmesh);

controls = new THREE.OrbitControls(camera, renderer.domElement);
}

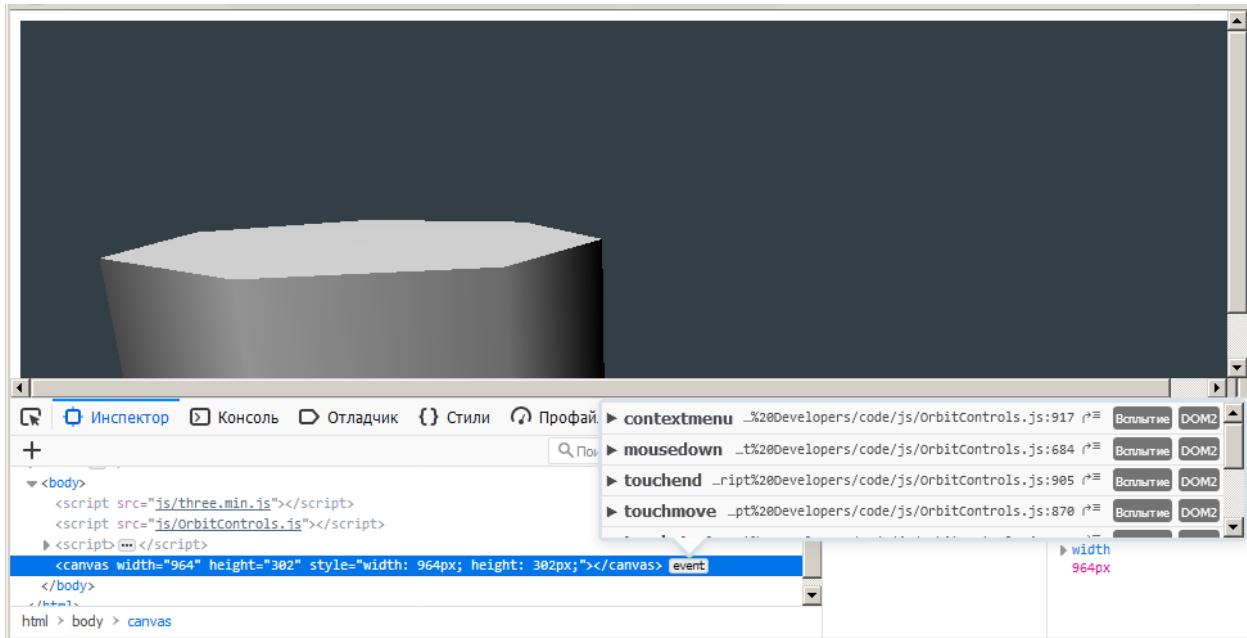
```

Зверніть увагу, що частина об'єктних змінних (scene, camera, renderer) створюються без використання ключового слова var – такі змінні, попри те, що створюються всередині функції init, стають глобальними та можуть бути використані поза її межами: вони знадобляться у функції animate.

Візуалізатор створюється без посилання на існуюче полотно, тому виклик WebGLRenderer створить нове полотно, яке розміщується у html-документі викликом document.body.appendChild(renderer.domElement) – у цьому легко переконатись, перейшовши у веб-браузері в режим налагодження.

Виклик window.addEventListener пов'язує подію зміни розмірів вікна браузера ('resize') із безіменною функцією, всередині якої визначаються нові розміри вікна (window.innerWidth та window.innerHeight), полотно масштабується до цих розмірів викликом renderer.setSize, для камери

встановлюється співвідношення сторін через властивість `aspect`, і нарешті після виконаної зміни параметрів виконується оновлення матриці проекціювання викликом `camera.updateProjectMatrix`.



Конструктор класу `OrbitControls` має один обов'язковий параметр – камеру, якою потрібно управляти. Другий (необов'язковий) параметр – елемент HTML, який використовується для обробників подій. За замовчуванням це весь документ цілком, однак, якщо потрібно, щоб елементи управління працювали від якогось певного елемента (наприклад, `<canvas>` – полотна), він вказується у другому параметрі.

Повний перелік властивостей класу `OrbitControls` можна знайти у документації – перелічимо лише деякі з них:

`autoRotate` – для автоматичного обертання встановіть значення даної властивості у `true`: якщо ця властивість має значення `false`, в циклі анімації потрібно викликати метод `update`;

`autoRotateSpeed` – визначає швидкість обертання камери навколо мети, якщо `autoRotate` встановлено у `true` (значення за замовчуванням дорівнює 2.0, що дорівнює 30 секундам на один оборот при 60 кадрах в секунду);

`enabled` – визначає, чи активний елемент управління;

`enableKeys` – включає або відключає управління від клавіатури;

`enablePan` – включає або відключає панорамування камерою;

`enableRotate` – включає або відключає горизонтальний і вертикальний поворот камери;

`enableZoom` – включає або відключає масштабування (наїзд – від'їзд) камери;

`screenSpacePanning` – визначає, як позиція камери змінюється під час панорамування: встановлено значення `true`, панорамування відбувається в просторі екрану, в іншому випадку камера повертається в площині, ортогональної до напрямку камери вгору;

`target` – точка фокусу елемента управління, навколо якої обертається камера.

Функція анімації містить виклик методу `update()`:

```
function animate()
{
    controls.update();
    renderer.render(scene, camera);
    requestAnimationFrame(animate);
}
```

Застосування класу `OrbitControls` надає можливість не лише огляду сцени, а й її масштабування.

### 3.8. Завантаження текстур та моделей

Текстури, як правило, являють собою зображення, які найчастіше створюються в сторонній програмі, такій як Photoshop або GIMP. Наприклад, для розміщення зображення на циліндрі з попереднього прикладу достатньо зробити наступні мінімальні модифікації:

– створити об'єкт класу `TextureLoader`:

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

– при визначенні матеріалу циліндра вказати, що властивість `map` містить завантажену за допомогою методу `load` класу `TextureLoader` текстуру:

```
var cylmaterial = new THREE.MeshLambertMaterial(  
    {map: loader.load('coke.png')});
```

Якщо текстура невелика за розміром, вона завантажиться досить швидко.



Доки текстура не буде завантажена, циліндр 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 наприклад, ми можемо дочекатися завантаження текстури, перш ніж створювати сітку і додавати її до сцени наступним чином:

```
//var cylmaterial = new THREE.MeshLambertMaterial({map: loader.load('coke.png')});
loader.load('coke.png', (texture) => {
    var cylmaterial = new THREE.MeshLambertMaterial({
        map: texture,
    });
    var cylmesh = new THREE.Mesh(cylgeometry, cylmaterial);
    cylmesh.position.set(0.9, -5, -6);
    scene.add(cylmesh);
});
```

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 сайтів цього не роблять. Наприклад, додамо до сцени куб, кожна грань якого міститиме власну текстуру:

```
var cubematerials = [
    new THREE.MeshBasicMaterial({map: loader.load(
        'https://farm1.staticflickr.com/854/42936929215_efa87c8a9a_b.jpg
    ')}),
    new THREE.MeshBasicMaterial({map: loader.load(
        'https://farm1.staticflickr.com/514/18832759790_bed1aeec8_b.jpg
    ')}),
    new THREE.MeshBasicMaterial({map: loader.load(
        'https://farm7.staticflickr.com/6140/5934453114_3675350a78_b.jpg
    ')})]
```

```
new THREE.MeshBasicMaterial({map: loader.load('https://farm3.static.flickr.com/2342/2269094999_6ac65c0947.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('https://live.staticflickr.com/1757/41821018565_614db58ddc_b.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('https://live.staticflickr.com/3103/2420240470_bc0bb7a260.jpg')}),
],
];

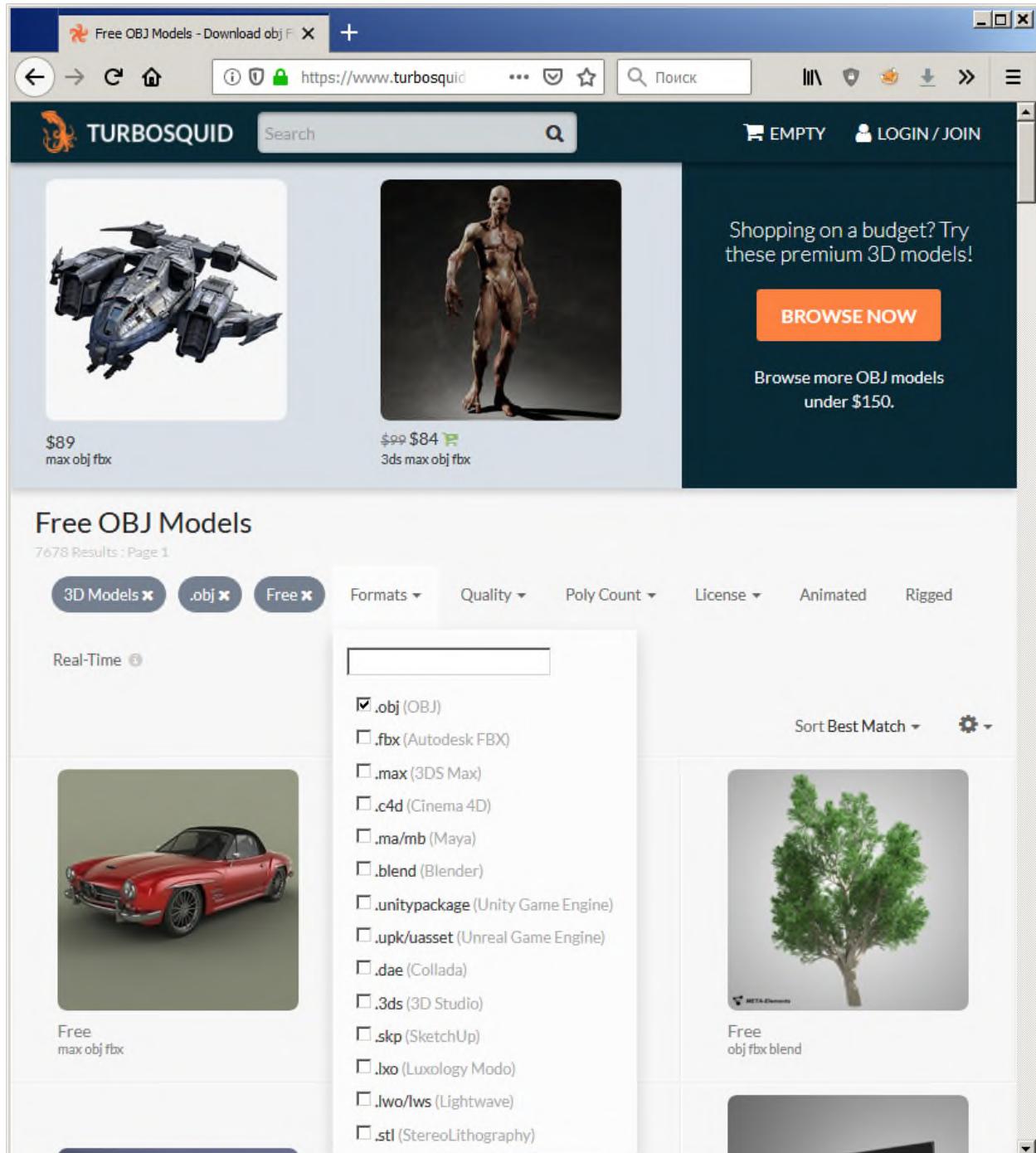
var cubegeometry = new THREE.CubeGeometry(3, 4, 3, 4);
var cubemesh = new THREE.Mesh(cubegeometry, cubematerials);
cubemesh.position.set(-1, -3, -2);
scene.add(cubemesh);
```



Однак слід зазначити, що за замовчуванням єдиною геометрією, яка

підтримує кілька матеріалів, є BoxGeometry / BoxBufferGeometry.

Для того, щоб завантажити модель за допомогою Three.js, її потрібно створити або узяти готову безкоштовну модель у форматі OBJ – наприклад, із сайту Turbosquid.com, та розмістити її у каталозі models.



Для завантаження моделі необхідний додатковий файл OBJLoader.js, який знаходитьться за шляхом three.js-dev/examples/js/loaders. У каталозі, в якому знаходитьсья файл OrbitControls.html, створимо новий файл OBJLoader.html та каталог js, до якого скопіюємо файл three.js-

dev/examples/js/loaders/OBJLoader.js.

```
<!DOCTYPE html>
<html>
    <head><title>OBJLoader</title></head>
    <body>
        <canvas id="Canvas"></canvas>
        <script src="three.js-dev/build/three.js"></script>
        <script src="js/OBJLoader.js"></script>
<script>
    var renderer = new THREE.WebGLRenderer({canvas: document.getElementById('Canvas'), antialias: true});
    renderer.setClearColor(0x333333);
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(window.innerWidth, window.innerHeight);

    var camera = new THREE.PerspectiveCamera(35, window.innerWidth / window.innerHeight, 0.1, 20000);

    var scene = new THREE.Scene();

    var lightOne = new THREE.AmbientLight(0xfffff, 0.5);
    scene.add(lightOne);

    var objLoader = new THREE.OBJLoader();
    objLoader.load("models/steyerdorf.obj", function(mesh){
        mesh.traverse(function(node){
            if(node instanceof THREE.Mesh){
                node.castShadow = true;
                node.receiveShadow = true;
            }
        });
        scene.add(mesh);
        mesh.position.set(-80, -50, -150);
        mesh.rotation.y = -Math.PI/4;
    });

    requestAnimationFrame(render);

    function render() {
        renderer.render(scene, camera);
        requestAnimationFrame(render);
    }
</script>
</body>
</html>
```

Для завантаження моделі парового локомотиву Steyerdorf використовується метод `load` класу `OBJLoader`, що має один обов'язковий та 3 необов'язкові аргументи:

`load(url, onLoad, onProgress, onError)`

`url` – рядок, що містить шлях/URL-адресу obj-файлу (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 виклик `load` із двома параметрами – шлях до моделі та безіменна функція, параметром якої є об'єкт із завантаженою моделлю `mesh`. Метод `traverse` класу `Object3D` у якості аргументу приймає безіменну функцію зворотного виклику, аргументом якої є об'єкт `node` класу `Object3D`. Дана функція викликається як для об'єкту, так й для всіх його нащадків – так, якщо викликати метод `traverse` для об'єкту `scene`, ми зможемо змінити всі розміщені на най об'єкти одним викликом.

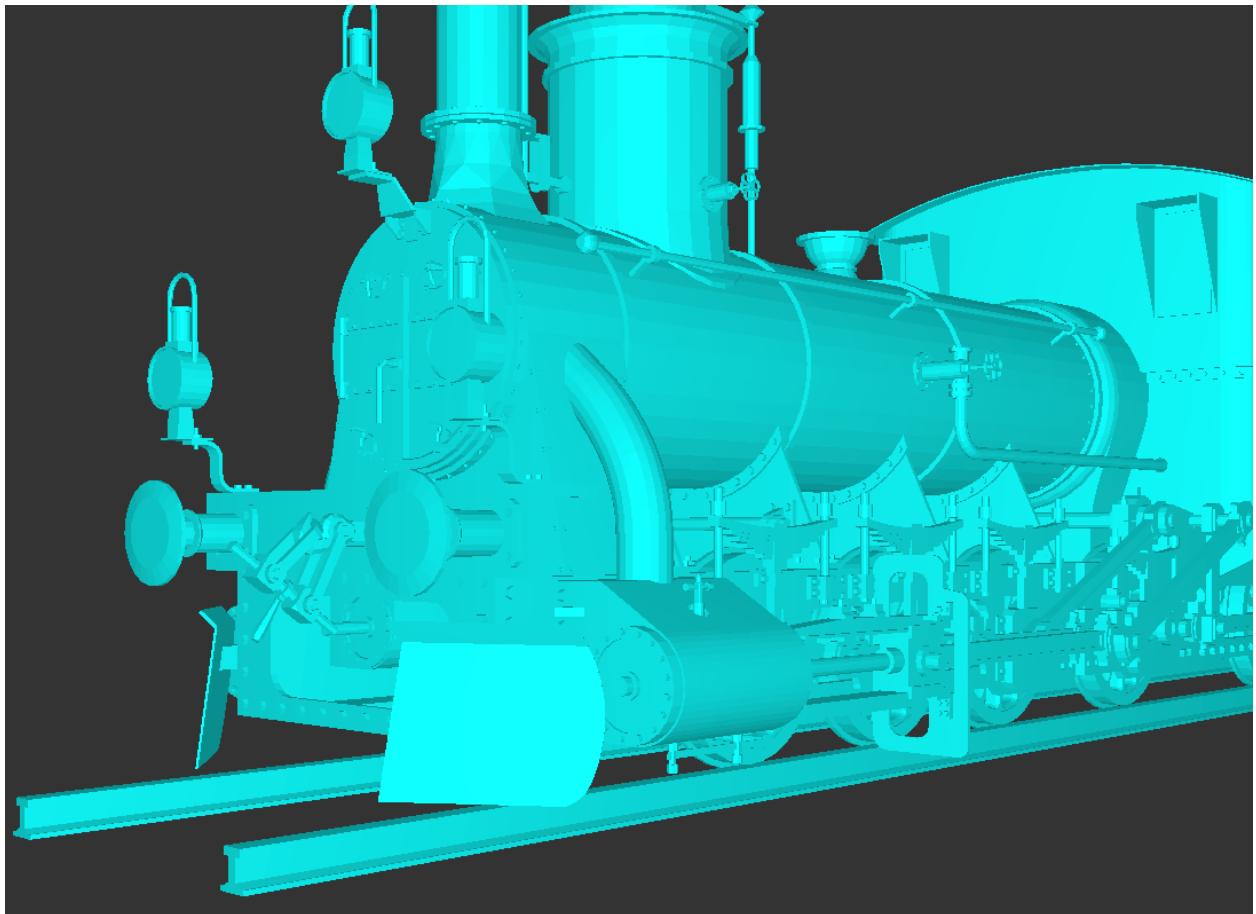
Метод `traverse` використовується нами для моделі – комплексного об'єкту, складеного із багатьох елементарних. Дляожної складової моделі за допомогою `instanceof` виконується перевірка, чи є відповідна складова об'єктом класу `Mesh`, і якщо так – активуються властивості приймання та відкидання тіней.

Далі модель розміщується на сцені за відповідними координатами та повертається на 45 градусів.



Додавши точкове джерело світла, можна побачити суттєво більше деталей:

```
var lightTwo=new THREE.PointLight(0xfffff, 0.5);  
scene.add(lightTwo);
```

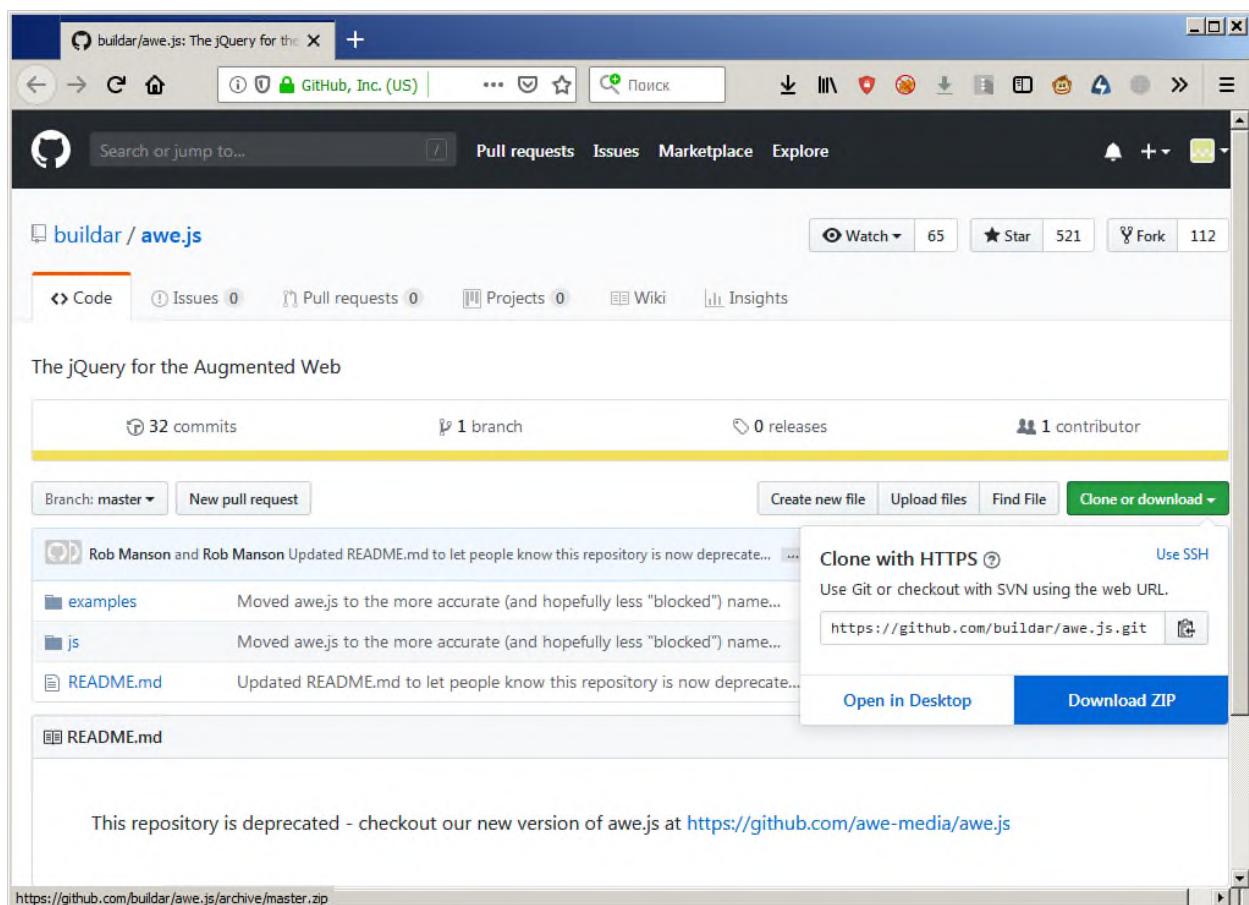


У Three.js існують й інші способи завантаження подібних моделей, у тому числі й ті, що не потребують окремого накладання текстур – за посиланням <https://github.com/mrdoob/three.js/tree/dev/examples/js/loaders> можна знайти більш ніж 40 завантажувачів для різних форматів зберігання моделей.

## 4 РОЗРОБКА КОМПЛЕКСНИХ AR-ПРОГРАМ ЗА ДОПОМОГОЮ AWE.JS

### 4.1. Налаштування та створення функції завантаження

Сьогодні awe.media та Daqri – провідні комерційні постачальники рішень із інтеграції віртуальної та доповненої реальності у веб-середовищі. Проте до того, як вони стали комерційними, бібліотеки awe.js та ARToolKit були провідними некомерційними AR-бібліотеками. Для того, щоб скористатись некомерційною версією першою із них, необхідно перейти за посиланням <https://github.com/buildar/awe.js> та завантажити архів проекту.



Після його розпакування отримаємо дерево каталогів наступного змісту:

```
awe.js-master
|
| README.md
|
| examples
|   geo_ar
|     awe.geo_ar.js
|     index.html
```

```

grift_ar
awe-rift-dependencies.js
awe.grift_ar.js
index.html

leap_ar
awe.leap_ar.js
index.html
leap.min-0.3.0.js

marker_ar
64.png
awe-jsartoolkit-dependencies.js
awe.marker_ar.js
awe_by_buildAR.png
index.html

js
awe-loader-min.js
awe-loader.js
awe-standard-dependencies.js
awe-standard-object_clicked.js
awe-standard-window_resized.js
awe-standard.js
awe-v8.js

```

Для зручності роботи створимо окремий каталог aweapp, до якого скопіюємо файл 64.png з awe.js-master\examples\marker\_ar та створимо порожній файл aweindex.html і каталог js. До останнього скопіюємо файли:

```

awe.js-master\js\awe-loader.js
awe.js-master\js\awe-standard.js
awe.js-master\js\awe-standard-dependencies.js
awe.js-master\js\awe-standard-object_clicked.js
awe.js-master\examples\marker_ar\awe-jsartoolkit-dependencies.js
awe.js-master\examples\marker_ar\awe.marker_ar.js

```

Каркас файлу aweindex.html складатиме код, у якому за допомогою тегу div визначатиметься контейнер як секція HTML-документу та завантажуватиметься основна бібліотека awe.js:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Russian Doll Boxes</title>

```

```

</head>
<body>
    <div id="container"></div>
    <script src="js/awe-loader.js"></script>
<script>
    ...
</script>
</body>
</html>

```

Розглянемо більш детально те, що буде розміщено між тегами `<script>` та `</script>`.

```

window.addEventListener('load', function(){
    window.awe.init({
        device_type: awe.AUTO_DETECT_DEVICE_TYPE,
        settings: {
            container_id: 'container',
            default_camera_position: {x: 0, y: 0, z: 0},
            default_lights: [
                {
                    id: 'point_light',
                    type: 'point',
                    color: 0xFFFFFFFF
                }
            ],
            ready: function(){
                awe.util.require([
                    {
                        capabilities: ['gum', 'webgl'],
                        files: [
                            ['js/awe-standard-dependencies.js', 'js/awe-standard.js'],
                            'js/awe-standard-object_clicked.js',
                            'js/awe-jsartoolkit-dependencies.js',
                            'js/awe.marker_ar.js'
                        ],
                        success: function(){
                            window.awe.setup_scene();
                            //to be continue...
                        } //success
                    },
                    []
                ]); //awe.util.require
            } //ready
        } //window.awe.init
    }); //load
});

```

Перш за все, нам необхідно створити безіменну функцію опрацювання події завантаження сторінки 'load' – фактично, у ній буде розміщений весь подальший код, що складається з єдиного виклику функції `window.awe.init. awe-loader.js`, починаючи з 767 рядка, містить опис функції `window`, у якій визначена функція `awe` (яка згодом перетворюється на конструктор одноіменного класу), в якій, в свою чергу, й визначена функція `init`. Її

параметром є об'єкт, що складається із трьох полів: `device_type`, `settings` та `ready`.

Поле `device_type` встановлюється у значення `awe.AUTO_DETECT_DEVICE_TYPE`. Серед інших можливих значень – `'ipad'`, `'iphone'`, `'android'`, `'pc'`.

Поле `settings` є об'єктом, що складається із трьох полів:

- `container_id` визначає місце у документі HTML, де буде розташовано DOM-елемент `awe_canvas`: полотно для малювання;
- `default_camera_position` визначає початкову позицію камери (якщо це налаштування не вказано, то у початок координат);
- `default_lights` є масивом, кожний об'єкт якого описує одне джерело світла параметрами `id` (ідентифікатор джерела світла), `type` (тип джерела світла) та `color` (колір освітлення). Доступні із Three.js типи джерел світла – `'area'`, `'directional'`, `'hemisphere'`, `'point'` та `'spot'`.

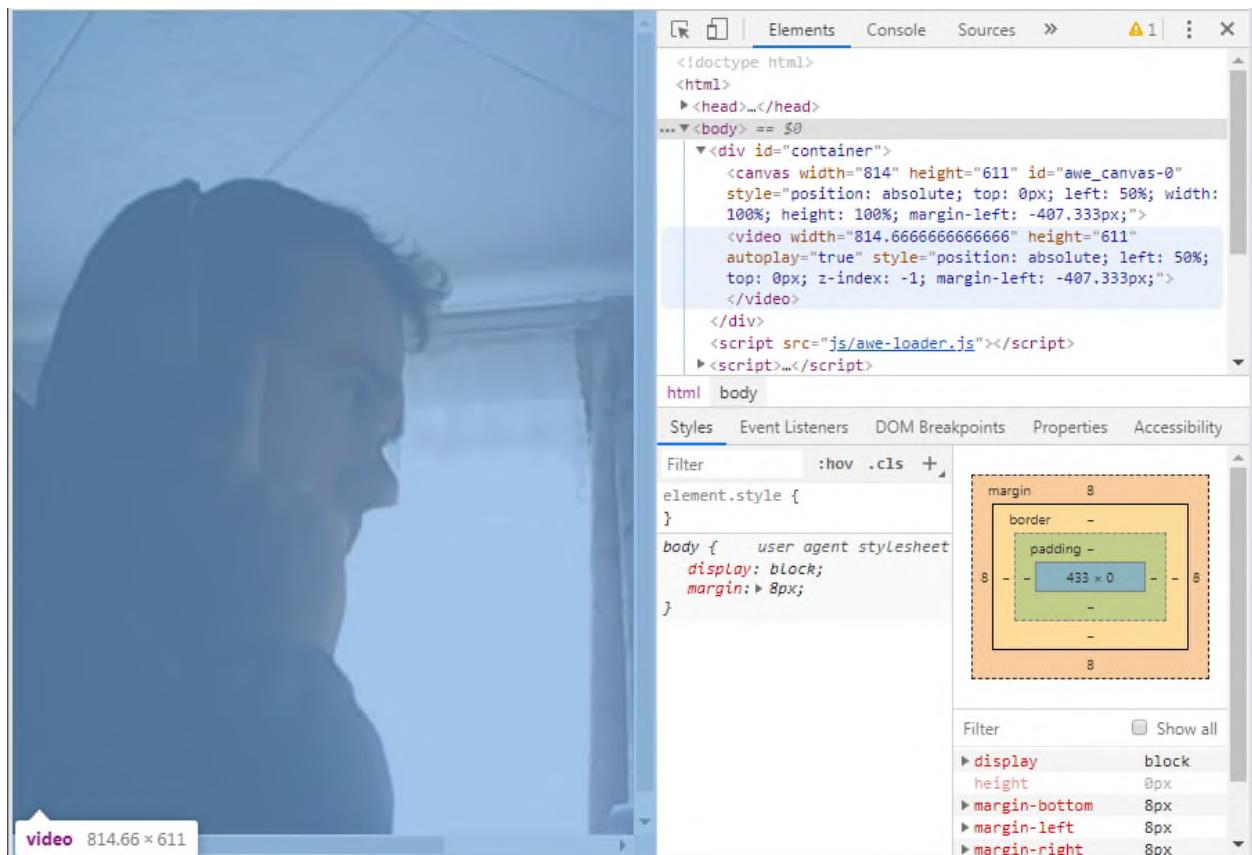
(Спробуйте додати четверте поле `fps: 30`, встановивши бажану кількість кадрів на секунду.)

Поле `ready` ініціалізується безіменною функцією, в якій виконується виклик функції `awe.util.require`. Її параметром є масив, що складається з одного об'єкту, який містить три поля:

- `capabilities` є масивом властивостей, що активуються: `'ajax'` (AJAX), `'geo'` (геолокація), `'lat'` (широта), `'lon'` (довгота), `'gyro'` (орієнтація пристрою у просторі), `'motion'` (рух пристрою), `'audio'` (веб-аудіо), `'gum'` (підтримка камери та мікрофону), `'webgl'` (WebGL), `'css3d'` (CSS3D), `'storage'` (підтримка локального зберігання даних), `'sockets'` (підтримка веб-сокетів);
- `files` є масивом імен файлів, розміщених у каталозі js (перші два файли утворюють свій власний масив);
- `success` є безіменною функцією, що містить виклик функції `window.awe.setup_scene` (і буде містити ще код). Метод `setup_scene`

визначає всі налаштування сцени, стандартні для Three.js – візуалізатор, джерела світла, співвідношення сторін сцени, її розмір та ін., та додає деякі інші, специфічні для доповненої реальності.

Ознакою того, що код вже працює, є не лише запит на використання камери – користуючись інспектором веб-браузера, можна побачити, що у контейнері тепер міститься не лише полотно, а й відеопоток:



## 4.2. Додавання ROI, проекцій та подій

Як можна зрозуміти з екранної копії у попередньому пункті, наступні дії будуть продовженням функції `success`. Після початкового налаштування сцени визначимо ROI (point of interest, дослівно – «цікаві точки» або місця) – у картографії це географічні об'єкти, відмічені на карті спеціальними позначками, а в термінах awe.js – місця на сцені, в яких буде знайдений маркер.

Всі елементи у awe.js позиціонуються в межах ROI. У awe.js можна рухати як ROI, так й елементи. Ми створимо одну ROI, яка буде розміщена усюди, де буде видимим визначений паперовий маркер:



Для створення POI застосуємо функцію `awe.pois.add()`:

```
awe.pois.add({
    id: 'marker',
    position: {x: 0, y: 0, z: 10000},
    visibility: false
});
```

Ідентифікатор створеного POI – 'marker'. Початкова позиція визначена як надзвичайно віддалена ( $z = 10000$ ) – це пов'язано із тим, що поки що POI не готовий до використання. POI (та всі пов'язані із ним елементи) є невидимим доти, доки не буде виявлено маркер.

Елементи, що додаються у POI, в `awe.js` називаються проекціями. Перша проекція, яку ми додаємо в нашу сцену, має назву 'largeBox' – це буде той самий великий прямокутний паралелепіпед, в якому за принципом матрьошки будуть вкладені всі інші. Ми додаємо його до визначеного POI, використовуючи функцію `awe.projection.add()`:

```
awe.projections.add({
    id: 'largeBox',
    geometry: {shape: 'cube', x: 180, y: 150, z: 180},
    position: {x: 0, y: 0, z: 0},
    rotation: {x: 90, z: 45},
    material: {color: 0xFFFFFFFF, opacity: 1.0, transparent: true,
               wireframe: false, fog: true}
}, {poi_id: 'marker'});
```

Існує досить багато варіантів об'єктів, які можна додати в якості проекцій, проте всі значення координат  $x$ ,  $y$  та  $z$  для позиціонування і повороту

об'єкту пов'язані з його POI, що POI визначається другим аргументом `awe.projections.add` як `{poi_id: 'marker'}`:

– `geometry` – параметри геометрії проекції у Three.js. Параметри, необхідні для кожного типу геометрії, відповідають параметрам, визначенім awe.js. Наприклад, `SphereGeometry` в Three.js буде представлений як `{shape: 'sphere', radius: 10}` в awe.js. Для тих, хто використовує останні версії Three.js, в поточній доступної версії awe.js `BoxGeometry` все ще позначається як `CubeGeometry`. Отже, для створення прямокутних паралелепіпедів ми використовуємо формат `{shape: 'cube', x: 20, y: 30, z: 5}` (незважаючи на назву, він не обов'язково повинен бути «кубом»). У `awe-standard.js` визначені такі типи геометрій: `'cube'` (у Three.js – `CubeGeometry`), `'sphere'` (`SphereGeometry`), `'cylinder'` (`CylinderGeometry`), `'lathe'` (`LatheGeometry`), `'octahedron'` (`OctahedronGeometry`), `'plane'` (`PlaneGeometry`), `'tetrahedron'` (`TetrahedronGeometry`), `'text'` (`TextGeometry`), `'torus'` (`TorusGeometry`), `'torusknot'` (`TorusKnotGeometry`), `'tube'` (`TubeGeometry`). Якщо геометрія явно не вказана, використовується `CubeGeometry`;

– `position` – встановлюються `x`, `y` і `z` координати елемента відносно його POI;

– `rotation` – обертання об'єкта у градусах навколо осей `x`, `y` і `z` відносно його POI;

– `material` – визначає властивості матеріалу проекції: `color` – колір у шістнадцятковому форматі (за замовчанням `0x404040`); `type` – тип матеріалу: `'phong'` (`MeshPhongMaterial`), `'lambert'` (`MeshLambertMaterial`), `'shader'` (`ShaderMaterial`), `'sprite'` (`SpriteMaterial`), `'sprite_canvas'` (`SpriteCanvasMaterial`) (за замовчанням матеріал – `MeshBasicMaterial`); `wireframe` – якщо `true`, показувати сітчасту геометрію; `side` – визначає, який бік об'єкту відкидає тіні: передній (`"front"`) чи задній (`"back"`) тощо;

– `texture` – визначає текстуру об'єкту: `texture: {path: 'file.png'}`;

– scale – встановлюється масштаб елемента по осіах x, y і z відносно його POI.

Друга проекція, прив'язана до визначеної POI – червоний куб меншого розміру, вкладений у попередній:

```
awe.projections.add( {
    id: 'mediumBox',
    geometry: {shape: 'cube', x: 90, y: 90, z: 90},
    position: {x: -5, y: -31, z: -5},
    rotation: {y: 45},
    material: {type: 'phong', color: 0xFF0000}
}, {poi_id: 'marker'});
```

Після того, як всі проекції визначені, виконується виклик функції `awe.events.add()`, якою й завершується функція `success`:

```
awe.events.add([
    {
        id: 'ar_tracking_marker',
        device_types: {pc: 1, android: 1},
        register: function(handler){
            window.addEventListener('ar_tracking_marker', handler, false);
        },
        unregister: function(handler){
            window.removeEventListener('ar_tracking_marker', handler, false);
        },
        handler: function(event){
            if (event.detail){
                if (event.detail['64']){
                    awe.pois.update({
                        data: {
                            visible: true,
                            position: {x: 0, y: 0, z: 0},
                            matrix: event.detail['64'].transform
                        }, where: {id: 'marker'}
                    });
                    awe.projections.update({
                        data: {visible: true},
                        where: {id: 'largeBox'}
                    });
                }
            }
            else{
                awe.pois.update({
                    data: {visible: false},
                    where: {id: 'marker'}
                });
            }
        }
    }
});
```

Параметром `awe.events.add` є масив об'єктів, кожен з яких визначає одну подію. Подія відстеження маркеру є об'єктом із наступними полями:

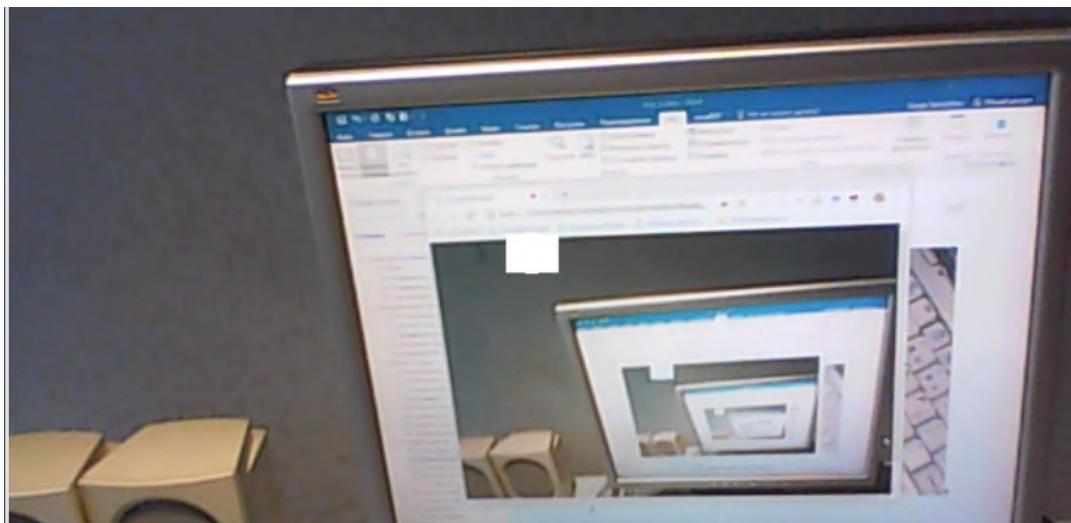
- `id` – ідентифікатор події;
- `device_types` – типи пристройів, до яких застосовується подія;
- `register` – функція додавання обробника `handler` події '`ar_tracking_marker`'. У `awe.js` саме обробник буде постійно відстежувати і виявляти, чи потрібно візуалізувати об'єкти. Якщо об'єкти візуалізуються і «натискаються», то він починає налаштовувати різні події, які можуть статися і які, в основному, створюватимуть нашу інтерактивну сцену;
- `unregister` – функція видалення обробника `handler` події '`ar_tracking_marker`' (скасування реєстрації події);
- `handler` – власне функція обробника події, яка буде запускатися після виявлення маркера.

Зараз обробник `handler` лише включає/виключає видимість об'єктів. Пізніше ми додамо до нього інші події, але зараз зосередимося лише на тому, щоб об'єкти були видимими, коли маркер виявлений, або щоб зробити їх невидимими, коли маркер не знайдений. Для цього на початку функції виконується перевірка, чи є у наявності будь-який маркер. Якщо маркер відсутній, то вся POI приховується викликом `awe.pois.update({data: {visible: false}, where: {id: 'marker'}})`:

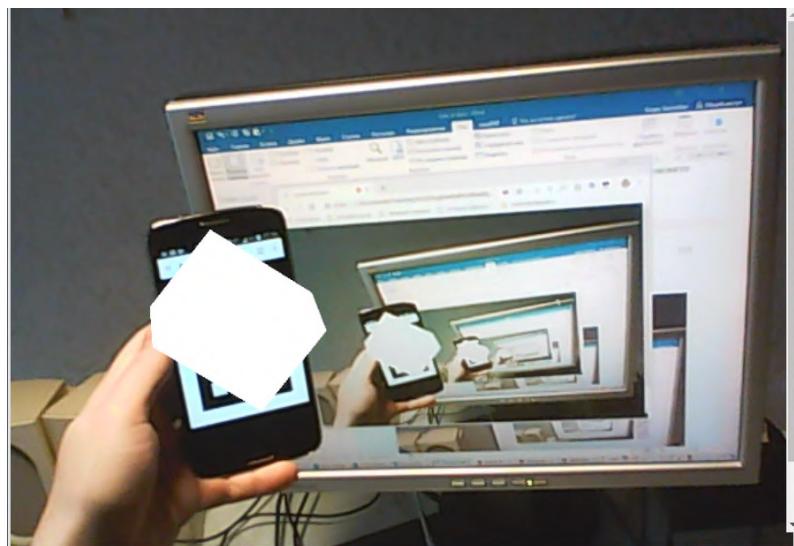
Якщо маркер знайдений, порівнюємо його із шуканим (64) і при співпадінні викликом `awe.pois.update` виконуємо переміщення POI у позицію фізичного паперового маркера (`position: {x: 0, y: 0, z: 0}`) та робимо його видимим (`visible: true`). Для того, щоб POI співпадав за розміром із фізичним маркером, виконуємо його масштабування наданням матриці проєціювання відповідного значення перетворення (`matrix: event.detail['64'].transform`).

Крім POI, видимими слід зробити й проекції: виклик `awe.projections.update` робить видимою (`visible: true`) першу з них (`id: 'largeBox'`).

Надання `awe.scene_needs_rendering` значення 1 сигналізує `awe.js` оновити сцену. На цьому тимчасово функцію `success` завершимо.



Після запуску `aweindex.html` на поток із камери накладається зображення білого об'єкту, що завжди знаходиться у центрі вікна до першого розпізнавання маркеру. Це – віддалений на 10000 по осі z ROI. Позбутися його можна, збільшивши дане значення до 30000.



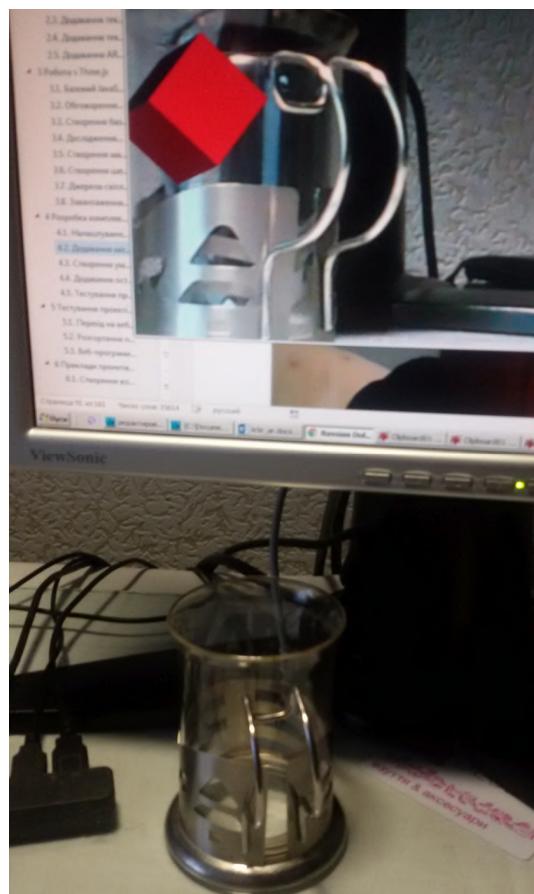
Маркер успішно розпізнається – поверх нього з'являється прямокутний паралелепіпед і, доки маркер відслідковується, змінює положення услід за ним. Проте, коли маркер виходить з поля зору камери, прямокутний паралелепіпед залишається на сцені, адже у блоці `else` умовного розгалуження в функції `handler` робиться невидимим лише ROI, а не пов'язані з ним проекції.



Для того, щоб позбутися білого прямокутного паралелепіпеда після виходу маркеру за межі поля зору камери, можна додати до умови `if (event.detail['64'])` блок `else` із наступним кодом:

```
awe.projections.update({data: {visible: false}, where: {id: 'largeBox'}});
```

Після цього білий прямокутний паралелепіпед дійсно щезатиме при нерозпізнаванні маркера – але тоді залишатиметься червоний куб ('mediumBox'). Тож припинимо ці експерименти та перейдемо до наступного пункту з метою додання логіки опрацювання створених об'єктів.



### 4.3. Створення умовних розгалужень

Досі створювані нами об'єкти доповненої реальності не були інтерактивними – всі їх зміни були запрограмовані завчасно та не залежали від дій користувача. Спробуємо визначити, наприклад, коли користувач обирає об'єкт 'largeBox', а коли – 'mediumBox'.

Для цього додамо у початок функції-обробника події 'load' (перед викликом window.awe.init) дві логічні змінні largeBox\_open та mediumBox\_open. Також змінимо поле settings, додавши у відповідний об'єкт поле fps – кількість кадрів на секунду:

```
window.addEventListener('load', function(){
    var largeBox_open = false;
    var mediumBox_open = false;

    window.awe.init({
        device_type: awe.AUTO_DETECT_DEVICE_TYPE,
        settings: {
            container_id: 'container',
            fps: 30,
            default_camera_position: {x: 0, y: 0, z: 0},
            default_lights: [
                {
                    id: 'point_light',
                    type: 'point',
                    color: 0xFFFFFFFF
                }
            ],
        },
    });
}
```

Повернемося до функції success та додамо після завершення awe.events.add новий обробник подій:

```
window.addEventListener('object_clicked', function(e) {
    switch (e.detail.projection_id) {
        case 'largeBox':
            if (!largeBox_open) {
                awe.projections.update({
                    data: {
                        animation: {duration: 1},
                        position: {y: 120}
                    },
                    where: {id: 'mediumBox'}
                });
            }
    }
});
```

```

        } );
    }
    else {
        awe.projections.update( {
            data: {
                animation: {duration: 1},
                position: {y: -31}
            },
            where: {id: 'mediumBox'}
        });
    }
    largeBox_open = !largeBox_open;
    break;
    case 'mediumBox':
        //...
        break;
    }
}, false);

```

Подія 'object\_clicked' відповідає натисканню кнопки миši на маркерному POI. Функція-обробник події викликається при натисканні на об'єкті e. Для конкретизації того, де саме було виконане натискання, аналізується e.detail.projection\_id – ідентифікатор проекції ('largeBox' та 'mediumBox').

Якщо натискання було виконано на проекції 'largeBox', перевіряється, чи встановлена логічна змінна largeBox\_open. Якщо ні, позиція другої проекції ('mediumBox') оновлюється протягом 1 секунди з метою створення ефекту появи червоного куба із білого прямокутного паралелепіпеду. Якщо так, виконується зворотна зміна у-координати червоного куба (ефект його зайду до білого прямокутного паралелепіпеду). Після виконання цих перевірок значення змінної largeBox\_open змінюється на протилежне.

Тестування надає можливість перевірити, як працює анімаційний ефект: після натискання білого прямокутного паралелепіпеду з'являється червона кнопка, яка після повторного натискання зникає.



Для завершення нашої кубоматрьошки одразу ж після додавання проекції з ідентифікатором 'mediumBox' створимо ще один кубічний об'єкт 'smallBox', менший за розміром, ніж 'mediumBox', та іншого кольору:

```
awe.projections.add({
    id: 'smallBox',
    geometry: {shape: 'cube', x: 50, y: 50, z: 50},
    position: {x: -5, y: -31, z: -5},
    rotation: {y: 45},
    material: {type: 'phong', color: 0xFF9889}
}, {poi_id: 'marker'});
```

Для того, щоб нова проекція висувалась із меншої, заповнимо блок опрацювання проекції 'mediumBox' аналогічно до попереднього:

```
case 'mediumBox':
    if (!mediumBox_open) {
        awe.projections.update({
            data: {
                animation: {duration: 1},
                position: {y: 190}
            },
            where: {id: 'smallBox'}
```

```

        });
    }
    else {
        awe.projections.update({
            data: {
                animation: {duration: 1},
                position: {y: -31}
            },
            where: {id: 'smallBox'}
        });
    }
    mediumBox_open = !mediumBox_open;
    break;
}

```

Таким чином, при натисканні на кожний об'єкт з нього висуватиметься новий, а при повторному натисканні – рухатимеся у зворотному напрямі.

Наведений код є не зовсім логічно правильним: так, він надає можливість прибрати другу проекцію ('mediumBox'), і на сцені залишатимуться видимими перша ('largeBox') та третя ('smallBox'). Зміна умови надасть можливість усунути це протиріччя.

Врешті-решт, можна додати блок й для опрацювання натискання на останній елемент. Наприклад:

```

case 'smallBox':
    window.open('https://www.google.com/search?q=%D0%A1%D0%B5%D0%BC%D0%B5%D1%80%D1%96%D0%BA%D0%BE%D0%B2');
    break;

```

Бібліотека awe.js підтримує далеко не всі браузери. Для перевірки, чи може вона працювати на обраному браузері, додається наступний код:

```

        case 'smallBox':
            window.open('https://www.google.com/search?q=%D0%A1%D0%B5%D0%BC%D0%B5%D1%80%D1%96%D0%BA%D0%BE%D0%B2');
            break;
        },
        },
        capabilities: [],
        success: function() {
            document.body.innerHTML = '<p>Try this application in the latest version of Chrome on a PC or Android device</p>';
        }
    ); //awe.util.require
}

```

До масиву об'єктів, що передає у функцію `awe.util.require`, додається об'єкт з полем `capabilities`, ініційованим порожнім масивом – він опрацьовується тоді, коли браузер не підтримує управління камерою і мікрофоном та WebGL. За цих ненайгарніших для роботи умов викликається функція `success`, яка до тіла документа додає текст про помилку.

#### 4.4. Завантаження моделей у Awe.js

Велика кількість вкладених функцій не полегшують сприйняття коду, тому зробимо його невеличкий рефакторинг. Для цього створимо новий файл awemodindex.html наступного змісту:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Awe.js model loader</title>
    </head>
    <body>
        <div id="container"></div>
        <script src="js/awe-loader.js"></script>
        <script src="my_model.js"></script>
    </body>
</html>
```

Замість того, щоб розташовувати програмний код між тегами `<script>` та `</script>`, він виноситься до окремого файлу `my_model.js`, розташованого в одному каталогу із `awemodindex.html`

Перша дія, що виконується у ньому – налаштування обробника події `'load'`. Функція `on_load` пов'язується із подією `'load'` викликом `window.addEventListener`:

```
window.addEventListener('load', on_load); //load
```

Єдина дія, що поки що виконується у функції `on_load` – налаштування бібліотеки `awe.js` за допомогою виклику `window.awe.init`:

```
function on_load()
{
    window.awe.init(
        device_type: awe.AUTO_DETECT_DEVICE_TYPE,
        settings: {
            container_id: 'container',
            fps: 30,
            default_camera_position: {x: 0, y: 0, z: 0},
            default_lights: [ {
```

```

        id: 'point_light',
        type: 'point',
        color: 0xFFFFFFFF
    } ]
},
ready: on_ready//ready
} ); //window.awe.init
}

```

Після того, як бібліотека успішно ініціалізована, викликається функція `on_ready`. Єдина дія, що виконується у ній – виклик `awe.util.require` з метою визначення двох випадків. У першому за наявності підтримки камери, мікрофону та WebGL виконується завантаження додаткових файлів бібліотеки, після чого викликається функція `on_success`. У другому – виводиться повідомлення про неможливість роботи у поточній програмно-апаратній конфігурації.

```

function on_ready()
{
    awe.util.require([
    {
        capabilities: ['gum', 'webgl'],
        files: [
            ['js/awe-standard-dependencies.js',           'js/awe-
standard.js'],
            'js/awe-standard-object_clicked.js',
            'js/awe-jsartoolkit-dependencies.js',
            'js/awe.marker_ar.js'
        ],
        success: on_success//success
    },
    {
        capabilities: [],
        success: function() {
            document.body.innerHTML = '<p>Try this application
in the latest version of Chrome on a PC or Android device</p>';
        }
    }
});

```

```

        }
    }
]) //awe.util.require
}

```

Дії із налаштування сцени, POI та проекцій виконуються у функції on\_success:

```

function on_success()
{
    window.awe.setup_scene();

    awe.pois.add({
        id: 'marker',
        position: {x: 0, y: 0, z: 30000},
        visibility: false
    });

    awe.projections.add({
        id: 'model',
        geometry: {path: 'models/female02.obj'},
        position: {x: 0, y: 0, z: 0},
        scale: {x: 2, y: 2, z: 2},
        material: {path: 'models/female02.mtl'},
        {poi_id: 'marker'});
    }

    awe.events.add([
        {
            id: 'ar_tracking_marker',
            device_types: {pc: 1, android: 1},
            register: function(handler){
                window.addEventListener('ar_tracking_marker',
handler, false);
            },
            unregister: function(handler){
                window.removeEventListener('ar_tracking_marker',
handler, false);
            },
        },
    ]);
}

```

```

    handler: on_handler
  } ] );
}

```

Із коду функції можна побачити, що єдина проекція, що розташовується на сцені – це завантажувана із каталогу model модель, яку описують два файли: obj-файл містить опис геометрії, mtl – матеріалу. Для відслідковування маркера за допомогою виклику awe.events.add додається функція on\_handler, яка й обробляє подію виявлення маркера:

```

function on_handler(event)
{
  if (event.detail){
    if (event.detail['64']){
      awe.pois.update({
        data: {
          visible: true,
          position: {x: 0, y: 0, z: 0},
          matrix: event.detail['64'].transform
        }, where: {id: 'marker'}
      });

      awe.projections.update(
        //data: {visible: true},
        position: {x: 0, y: 0, z: 0},
        where: {id: 'model'}
      );
    }
    else {
      awe.pois.update({
        data: {visible: false},
        where: {id: 'marker'}
      });

      awe.projections.update(
        position: {y: -1000000},
        where: {id: 'model'}
      );
    }
  }
}

```

```
        }  
        awe.scene_needs_rendering = 1;  
    }  
}
```

На відміну від простих проекцій, виклик `awe.projections.update` із атрибутом `{visible: false}` не приховує модель цілком – замість цього її достатньо вивести за межі камери встановленням віддаленої позиції.

Результат виглядає досить пристойно:



Бібліотека awe.js використовує JSARToolKit для розпізнавання маркерів, що надає можливість використовувати велику кількість підтримуваних JSARToolKit типів маркерів. Так, чи розпізнано маркер з файлу 64.png, перевірялось простим зверненням до масиву event.detail за символічним індексом '64'. Цей та інші файли можна знайти у репозитарії за посиланням <https://github.com/kig/JSARToolKit/blob/master/demos/markers>

У такий спосіб можна одночасно розпізнавати до 100 маркерів, чого цілком достатньо для невибагливих користувачів.

## 5 ДОПОВНЕНА РЕАЛЬНІСТЬ НА МОБІЛЬНИХ ПРИСТРОЯХ

### 5.1 Налаштування параметрів консолі 8th Wall

8th Wall – один із комерційних постачальників AR-послуг, розпочати роботу з яким можна й у досить функціональному умовно безкоштовному режимі.

8th Wall Web – JavaScript-бібліотека, що реалізує технологію одночасної локалізації та картографування (Simultaneous Localization and Mapping - SLAM), яка широко використовується в безпілотних автомобілях, безпілотних літаючих засобах, автономних підводних апаратах, планетоходах, побутових роботах тощо.

8th Wall Web легко інтегрується із такими бібліотеками, як A-Frame та three.js, а також Babylon.js (<https://www.babylonjs.com/>) і Amazon Sumerian (<https://aws.amazon.com/sumerian/>).

Так само, як й awe.js, 8th Wall Web вимагає використання браузера за протоколом HTTPS для доступу до веб-камери. Для мобільних браузерів обов'язкова підтримка:

- WebGL (canvas.getContext('webgl') || canvas.getContext('webgl2'));
- getUserMedia (navigator.mediaDevices.getUserMedia);
- deviceorientation (window.DeviceOrientationEvent);
- Web-Assembly/WASM (window.WebAssembly).

Для початку роботи із 8th Wall Web необхідно створити за посиланням <https://console.8thwall.com/sign-up> обліковий запис, обравши пункт «Web Developer» (два інші – «AR Camera» та «XR Developer» призначені відповідно для створення доповненої реальності на основі готових моделей та розробки за допомогою Unity).

Після SMS-підтвердження необхідно уважно прочитати умови використання бібліотеки та погодитись із ними (зверніть особливу увагу на 5 пункт) та обрати базовий тип облікового запису (фінансовий план для локальної розробки із обмеженням кількості переглядів протягом місяць).

## Sign up for free

First Name \*

Last Name \*

Email \*

I would like to:



**Selected**

Create custom AR experiences for the mobile web using JavaScript and WebGL with [Web Developer](#)



Instantly create AR cameras for the mobile web by uploading 3D models with [AR Camera](#)



Build mobile AR apps using Unity and deploy to iOS and Android simultaneously with [XR Developer](#)

Company Size

Password \*

Password Confirmation \*

I'm not a robot   
[Privacy](#) - [Terms](#)

**Get Started**



WORKSPACES [Web Developer](#)

DASHBOARD

QUICK START

TEAM

PLAN + BILLING

WEB DEVELOPER DASHBOARD

**Web App Views**

You've had no recent views.

0  1,000 2,000 10,000 [Web Developer Pro Plan](#)

**My Web Applications**

[+ Create a new Web App](#)

Make sure to review the Quick Start Guide and Web Documentation 

[Quick Start](#) [Documentation](#)

Для початку розробки програми у консолі веб-розробника необхідно обрати пункт «Create a new Web App» та увести ім'я створюваної програми (наприклад, WebARTest). Після цього у консолі з'являться відомості про створювану програму, з яких необхідним для подальшої роботи є ключ програми (поле «My App Key»).

## CREATE NEW WEB APPLICATION

For Web Applications, the application name should be descriptive but short and only include letters, numbers, "-" and "\_". It should be unique. It may only be between 4 and 128 characters in length. You cannot change the name of the application later, but you can always create another application.

Application Name \*

WebARTest

**Create** Cancel

WebARTest 0 Views

STATUS **LIVE**

AVERAGE DAILY VIEWS 0.0

ALLOWED ORIGINS None connected

**Enable public access**

MY APP KEY jRPuWR0Fd4poFvFg... **Copy**

IMAGE TARGETS **NEW**

Add an image target

Total Daily

Total Views

May 30 May 31 Date

Для використання бібліотеки необхідно вказати згенерований ключ замість «XXXXXX» у відповідному HTML-файлі програми:

– для A-Frame:

```
<script async src="https://apps.8thwall.com/xrweb?appKey=XXXXXX">
</script>
```

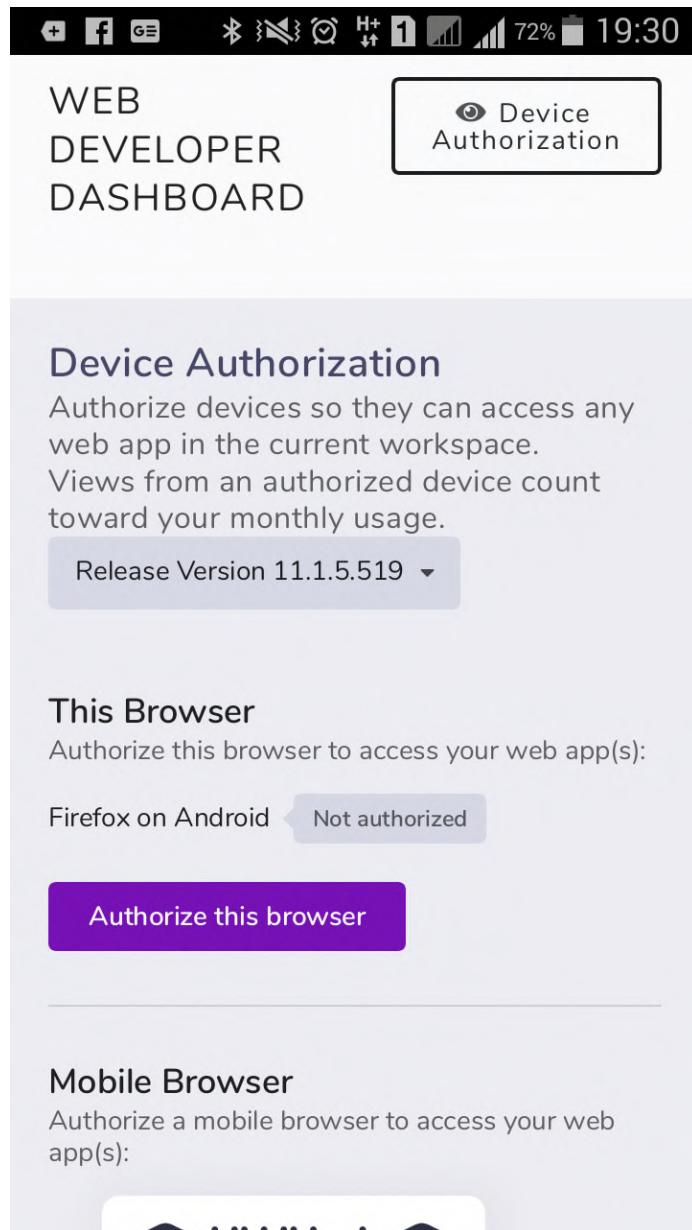
– для three.js:

```
<script defer src="https://apps.8thwall.com/xrweb?appKey=XXXXXX">
</script>
```

Найпростіший спосіб перегляду створюваної програми на мобільному пристрої – обрати для неї пункт «Enable public access», що потребує зміни тарифного плану з безкоштовного Basic на платний, тому підемо іншим шляхом, авторизувавши мобільний пристрій для використання режиму розробника.

Авторизація пристрою надає можливість виконувати на ньому веб-програми з використанням 8th Wall. Кількість авторизованих пристрой не обмежена, але кожен пристрій має бути авторизованим окремо, а перегляди веб-програми на кожному пристрої враховуються у щомісячний ліміт використання програми – 1000 переглядів.

Зauważимо, що авторизувати можна будь-який як мобільний браузер, відкривши консоль розробника у ньому, так й скануванням QR-коду.



## 5.2 Спільна робота 8th Wall та A-Frame

Для додавання підтримки 8th Wall Web у проекти з A-Frame необхідно замість стандартної версії бібліотеки A-Frame включити модифіковану 8-

Frame:

```
<script src="//cdn.8thwall.com/web/aframe/8frame-0.9.0.min.js">
</script>
```

Номери версії бібліотек співпадають – щоб дізнатися актуальний, завітайте до <https://aframe.io>

Посилання на ключ бажано розмістити між тегами <head>...</head>:

```
<script async src="https://apps.8thwall.com/xrweb?appKey=XXXXXX">
</script>
```

(де XXXXXX – значення ключа програми)

Функціональність 8th Wall Web активується додаванням параметру xrweb до тегу a-scene:

```
<a-scene xrweb>
```

Якщо відслідковування положення не потрібне (роздізнаються тільки зображення), то достатньо вказати

```
<a-scene xrweb="disableWorldTracking: true">
```

У репозитарії <https://github.com/8thwall/web/>, крім численних прикладів, можна знайти бібліотеку XR Extras для розширеного контролю виконання програми через наступні параметри тегу a-scene:

xrextras-almost-there – визначає, чи підтримує пристрій або браузер 8th Wall Web, та надає інструкції із запуску програми;

xrextras-loading – відображає процес завантаження бібліотеки та авторизацію доступу до камери;

xrextras-runtime-error – відображає повідомлення про помилки;

xrextras-tap-recenter – при натисканні розміщує сцену всередині вікна огляду.

Для підключення бібліотеки можна скористатись її локальним варіантом або посиланням на сайт 8th Wall:

```
<script src="//cdn.8thwall.com/web/xrextas/xrextas.js">
</script>
```

Продемонструємо роботу 8th Wall Web, внесши модифікації до сцени, розглянутої у другому розділі:

```

<!DOCTYPE html>
<html>
    <head>
        <script
src="https://cdn.8thwall.com/web/aframe/8frame-0.9.0.min.js">
</script>
        <script
src="https://cdn.8thwall.com/web/xrextras/xrextras.js"></script>
        <script async
src="https://apps.8thwall.com/xrweb?appKey=XXXXXX"></script>
    </head>
<body>
    <a-scene xrweb xrextras-tap-recenter xrextras-almost-there
xrextras-loading xrextras-runtime-error>
        <a-torus position="0 0.5 0" color="green" radius="0.5">
</a-torus>
        <a-plane
src="https://raw.githubusercontent.com/aframevr/sample-
assets/master/assets/images/illustration/758px-
Canestra_di_frutta_(Caravaggio).jpg"
            width="3.5" height="3.5" rotation="-55 0 0"
position="0 -1 0" color="purple"></a-plane>
        <a-cylinder color="yellow" height="2" radius="0.05"
position="0 0 0"></a-cylinder>
        <a-cylinder color="blue" height="2" radius="0.05"
position="1 0 0"></a-cylinder>
        <a-torus-knot color="orange" radius="0.5" position="1 0.5
0"></a-torus-knot>
        <a-plane
src="https://raw.githubusercontent.com/aframevr/sample-
assets/master/assets/images/uvgrid/UV_Grid_Sm.jpg" width="2.5"
height="1.5" position="0 1 -1"></a-plane>
        <a-text value="Welcome to WebAR" color="black" width="3"
position="-1 0.5 -1"></a-text>
        <a-camera position="0 5 5"></a-camera>
    </a-scene>

```

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

Сцена завжди буде видимою та прив'язаною для певної точки простору, зміна якої виконується натисканням на екран.

