TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

MÔN ĐỒ HỌA MÁY TÍNH

XÂY DỰNG ỨNG DỤNG 3D TĨNH VÀ ĐỘNG DỰA VÀO HTML5 VÀ WEBGL

- **Nhóm code dạo:**

1612835 Bùi Trọng Xuyến
1612823 Trần Thanh Vũ
1612908 Đặng Tiến Dũng
1612831 Bùi Thúy Vy
1612165 Nguyễn Đào Vinh Hải

TP. HCM, ngày 01/01/2019

LỜI TỰA

WebGL (thư viện đồ họa web) là tiêu chuẩn mới cho đồ họa 3D trên web, được thiết kết để hiển thị độ họa 3D và đồ họa 3D tương tác. Hướng dẫn này bắt đầu bằng phần giới thiệu cơ bản về WebGL, phần tử canvas của HTML5, sau đó là phần hướng dẫn từng bước để chạy được WebGL, sử dụng mã nguồn mở Three.js và cuối cùng là phần giới thiệu về ứng dụng nhà "3D house".

Những hướng dẫn này được dịch từ các loại sách tiếng anh, các trang hỗ trợ học lập trình web nên sẽ có những thuật ngữ (từ ngữ) được phiên dịch không xác nghĩa mà chỉ phù hợp với hoàn cảnh hiện tại.

Mục lục

A. GIỚI THIỆU

- **WebGL là gì?**

WebGL (Thư viện đồ họa Web) là tiêu chuẩn mới cho đồ họa 3D trên Web, Nó được thiết kế cho mục đích hiển thị đồ họa 2D và đồ họa 3D tương tác. Nó được lấy từ thư viện ES 2.0 của OpenGL, đây là API 3D cấp thấp dành cho điện thoại và các thiết bị di động khác. WebGL cung cấp chức năng tương tự ES (hệ thống nhúng) và hoạt động tốt trên phần cứng đồ họa 3D hiện đại.

Nó là một API JavaScript có thể được sử dụng với HTML5. Mã WebGL được viết trong thẻ <canvas> của HTML5. Đây là một đặc điểm kỹ thuật cho phép các trình duyệt Internet truy cập vào Bộ xử lý đồ họa (GPU) trên các máy tính mà chúng được sử dụng.

- **Người phát triển**

Một kỹ sư phần mềm người Mỹ gốc Serbia tên **Vladimir Vukicevic** đã làm nền tảng và lãnh đạo việc tạo ra WebGL

- Năm 2007, Vladimir bắt đầu làm việc trên nguyên mẫu OpenGL cho phần tử Canvas của tài liệu HTML.
- Vào tháng 3 năm 2011, tập đoàn Kronos đã tạo ra WebGL.

- **Renderer**

Kết xuất là quá trình tạo hình ảnh từ một mô hình bằng các chương trình máy tính. Trong đồ họa, một cảnh ảo được mô tả bằng cách sử dụng thông tin như hình học, quan điểm, kết cấu, ánh sáng và bóng, được truyền qua một chương trình kết xuất. Đầu ra của chương trình kết xuất này sẽ là một hình ảnh kỹ thuật số.

Có 2 loại kết xuất:

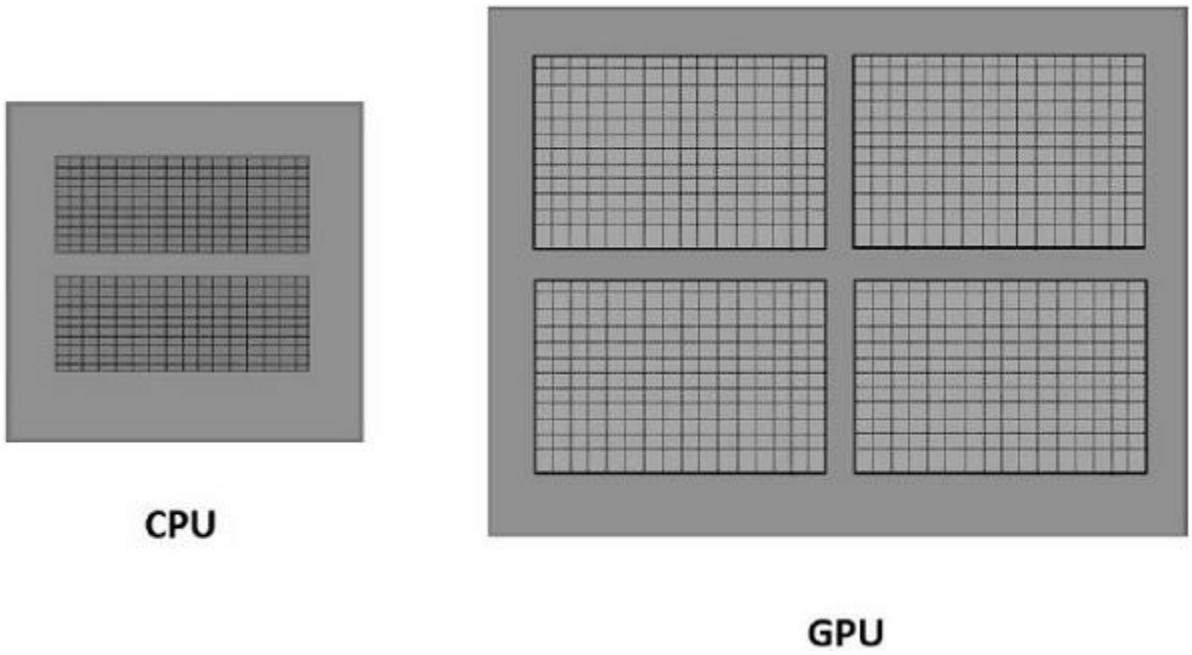
- Kết xuất phần mềm – tất cả các tính toán kết quả kết xuất được thực hiện với sự trợ giúp của CPU.
- Kết xuất phần cứng – tất cả tính toán đồ họa được thực hiện bởi GPU (đơn vị xử lý đồ họa).

Kết xuất có thể được thực hiện cục bộ hoặc từ xa. Nếu hình ảnh được kết xuất quá phức tạp, thì việc kết xuất được thực hiện từ xa trên một máy chủ chuyên dụng có đủ tài nguyên phần cứng để hiển thị cảnh phức tạp đó. Nó cũng được gọi là kết xuất dựa trên máy chủ. Kết xuất cũng có thể được thực hiện cục bộ bởi GPU. Nó được gọi là kết xuất dựa trên máy khách.

WebGL tuân theo cách tiếp cận kết xuất dựa trên máy khách để hiển thị cảnh 3D. Tất cả quá trình xử lý cần thiết để có thể hiển thị được hình ảnh được thực hiện cục bộ bằng phần cứng đồ họa máy khách.

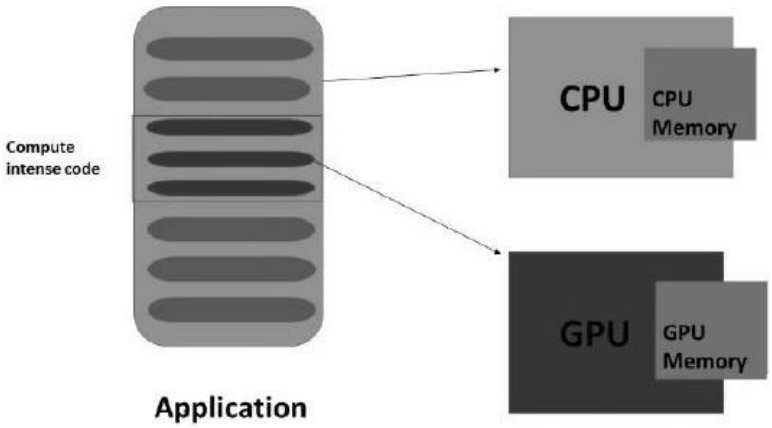
- **GPU**

Theo NVIDIA, GPU là "bộ xử lý chip đơn với tích hợp biến đổi, ánh sáng, thiết lập/cắt hình tam giác và các công cụ kết xuất có khả năng xử lý tối thiểu 10 triệu đa giác mỗi giây". Không giống như bộ xử lý đa lõi với một vài lõi được tối ưu hóa để xử lý tuần tự, GPU bao gồm hàng ngàn lõi nhỏ xử lý khối lượng công việc song song hiệu quả. Do đó, GPU tăng tốc việc tạo hình ảnh trong bộ đệm khung (một phần ram chứa dữ liệu khung hoàn chỉnh) dành cho đầu ra cho màn hình.



- **Tăng tốc GPU**

Trong điện toán tăng tốc GPU, ứng dụng được tải vào CPU. Bất cứ khi nào nó gặp một phần mã **chuyên sâu tính toán** , thì phần mã đó sẽ được tải và chạy trên GPU. Nó cung cấp cho hệ thống khả năng xử lý đồ họa một cách hiệu quả.



GPU sẽ có một bộ nhớ riêng và nó chạy nhiều bản sao của một phần nhỏ mã cùng một lúc. GPU xử lý tất cả dữ liệu trong bộ nhớ cục bộ của nó, không phải bộ nhớ trung tâm. Do đó, dữ liệu cần được xử lý bởi GPU nên được tải / sao chép vào bộ nhớ GPU và sau đó được xử lý.

Trong các hệ thống có kiến trúc trên, cần giảm chi phí liên lạc giữa CPU và GPU để xử lý các chương trình 3D nhanh hơn. Đối với điều này, chúng tôi phải sao chép tất cả dữ liệu và giữ nó trên GPU, thay vì liên lạc với GPU nhiều lần.

- **Trình duyệt hỗ trợ**

Để kiểm tra trình duyệt có hỗ trợ WebGl, chỉ cần truy cập website – https://get.webgl.org

Nếu:

- Your browser supports WebGL thì nghĩa là trình duyệt của bạn đã được hỗ trợ.
- Oh no! We are sorry, but your browser does not seem to support WebGL thì nghĩa là trình duyệt của bạn chưa được hỗ trợ WebGL, bạn cần phải bật Webgl lên tùy theo trình duyệt mà có các cách kích hoạt khác nhau.

- **Ưu điểm của WebGL**
  - Ứng dụng WebGL được viết bằng JavaScript nên các ứng dụng này có thể tương tác trực tiếp với các phần tử HTML, ngoài ra ta có thể thêm các thư viện JavaScript và các công nghệ HTML để hỗ trợ cho ứng dụng.
  - WebGL cũng hỗ trợ cho nền tảng di động.
  - WebGL là mã nguồn mở.
  - WebGL sử dụng JavaScript để code vì vật nó được hỗ trợ tự động quản lý bộ nhớ.
  - WebGL không cần thiết phải biên dịch để chạy.
  - Dễ dàng thiết lập và chạy, chỉ cần một editer và trình duyệt.

B. HTML5 VÀ API MỚI

1. Giới thiệu

HTML5 được hỗ trợ hầu hết các trình duyệt. Nó là một tập hợp các tính năng đặc biệt, như ta có thể tìm thấy hỗ trợ cho một số phần đặc trưng như canvas, video hoặc location.

HTML5 hỗ trợ tất cả các hình thức kiểm soát từ HTML4, nhưng nó cũng bao gồm điều khiển đầu vào mới. Một số trong số này là quá hạn bổ sung như các thanh trượt và date pickkers.

"Nâng cấp" lên HTML5 đơn giản chỉ bằng việc thay đổi thẻ DOCTYPE.

```
<!DOCTYPE html>
```

Nâng cấp lên DOCTYPE HTML5 sẽ không phá vỡ cấu trúc tài liệu của bạn, bởi lẽ các thẻ lỗi thời trước đây được định nghĩa trong HTML4 vẫn được vẽ ra trong HMTL5. Nhưng nó cho phép bạn sử dụng và xác nhận các thẻ mới như <article> <section>, <header>, và <footer>,...

2. Các API mới

- **Web Storage (DOM Storage)**

HTML5 cung cấp một tính năng lưu trữ dữ liệu tại client với giới hạn dung lượng lớn hơn hẳng cookie. Tính năng này được gọi là web storage và được chia thành hai đối tượng là localStorage và sessionStorage.

```
interface Storage {
        readonly attribute unsigned long length;
        DOMString? key(unsigned long index);
        getter DOMString getItem(DOMString key);
        setter creator void setItem(DOMString key, DOMString value);
        deleter void removeItem(DOMString key);
        void clear();
}
```

Local Storage và session Storage được tạo ra từ interface Storage, bạn sử dụng hai đối tượng này trong javaScript thông qua hai biến được tạo sẵn là window.localStorage và window.sessionStorage. sessionStorage giới hạn trong một cửa sổ hoặc thẻ của trình duyệt. Một trang web được mở trong hai thẻ của cùng một trình duyệt cũng không thể truy xuất dữ liệu lẫn nhau. Như vậy, khi bạn đóng trang web thì dữ liệu trong sessionStorage hiện tại cũng bị xóa theo. Còn localStorage có thể truy xuất lẫn nhau giữa các cửa sổ trình duyệt. Dữ liệu sẽ được lưu giữ không giới hạn thời gian.

Ví dụ 2-1. Hello.html

```
<!DOCTYPE html>
<html>
<body>
        <script type = "text/javascript">
                sessionStorage.myVariable = "Hello. ";
```

```
                localStorage.myVariable = “Nice to meet you!”;
                document.write(sessionStorage.myVariable);
                document.write(localStorage.myVariable);
        </script>
</body>
</html>
```

Storage event, event được kích hoạt mỗi khi storageArea bị thay đổi. Tức là khi bạn mở nhiều cửa sổ trình duyệt truy xuất đến một domain, nếu bạn thay đổi một đối tượng storage bên cửa sổ này thì các cửa sổ còn lại sẽ được kích hoạt event “storage”, còn cửa sổ hiện tại sẽ không xảy ra gì cả.

```
interface StorageEvent: Event {
        readonly attribute DOMString key;
        readonly attribute DOMString? oldValue;
        readonly attribute DOMString? newValue;
        readonly attribute DOMString url;
        readonly attribute Storage? storageArea;
}
```

Các storage có thể không đáp ứng yêu cầu của bạn, vì vậy bạn có thể thêm một vài phương thức cho bạn định nghĩa vào. Ví dụ ta sẽ thêm phương thức search() để lọc ra các giá trị chứa từ khóa cần tìm kiếm.

```
Storage.prototype.search = function(keyword) {
        var array = new Array();
        var re = new RegExp(keyword, “gi”);
        for (var i = 0; i < this.length; i++) {
                var value = this.getItem(this.key[i]);
                if (value.search(re) > -1) array.push(value);
        }
        return array;
}
```

- **Web SQL Database**

Là một công nghệ kết hợp giữa trình duyệt và SQLite để hỗ trợ việc tạo và quản lý database ở phía client. Các thao tác với database sẽ được thực hiện bởi JavaScript và sử dụng các câu lệnh SQL để truy vấn dữ liệu.

Open Database: mở một database có sẵn hoặc tạo mới nếu nó chưa tồn tại. Phương thức được khai báo như sau:

```
Database openDatabase(
        in DOMSting name,
        in DOMString version,
        in DOMString displayName,
        in unsigned long estimatedSize,
        in optional DatabaseCallback creationCallback;
);
```

Transaction: cung cấp khả năng rollback khi một trong những câu lệnh bên trong nó thực thi bất bại. Nghĩa là nếu bất kì một lệnh SQL nào thất bại, mọi thao tác thực hiện trước đó trong transaction sẽ bị hủy bỏ và database không bị ảnh hưởng gì cả.

```
void transaction (
        in SQLTransactionCallback callback,
        in optional SQLTransactionErrorCallback errorCallback;
        in optional SQLVoidCallback successCallback
```

```
);
```

Excute SQL: là phương thức duy nhất để thực thi một câu lệnh SQL trong Web SQL. Nó được sử dụng cho cả mục đích đọc và ghi dữ liệu:

```
void excuteSql (
        in DOMString sqlStatement,
        in optional ObjectArray arguments,
        in optional SQLStatementCallback callback;
        in optional SQLStatementErrorCallback errorCallback
);
```

- **Web Worker**

Hiện nay để giải quyết vấn đề đa luồng xử lý nhiều công việc, một API mới dành cho JavaScript là Web Worker.

Điểm hạn chế của Web Worker là không thể truy xuất được thành phần trên DOM, và cả các đối tượng window, document, hay parent. Mã lệnh các công việc cần thực thi cũng phải được cách ly trong một tập tin script. Ví dụ

Ví dụ 2-2. Chương trình Hello (js + html)

Mytask.js:

```
this.onmessage = function(event) {
        var name = event.data;
                postmessage("Hello " + name);
};
```

Test.html:

```
<!DOCTYPE html>
<body>
<input type = "text" id = username" value = "2" />
<br/>
<buttom id  = "buttom1">Submit</buttom>
<script>
        worker = new Worker("Mytask.js");
        worker.onmessage = function(event) {
                alert(event.data);
        }
        document.getElementById("buttom1").onclick = fucntion() {
                var name = document.getElementById("username").value;
                worker.postMessage(name);
        };
</script>
</body>
</html>
```

- **Tạo chuyển động với WindowAnimationTiming API**

setTimeout vaf setInterval: cách truyền thống liên tục update và draw sau những khoảng thời gian xác định thông qua phương thức setTimeout() và setInterval(). Mỗi lần gọi, một frame mới sẽ được tạo ra và đè lên màn hình cũ.

WindowAnimationTiming: Việc xác định thời điểm cập nhập frame sẽ được tự động chọn giá trị thích hợp nhất.

```
interface WindowAnimationTiming {
        long requestAnimationFrame(FrameRequestCallback callback);
        void cancelAnimationFrame(long handle);
```

```
};
Window implements WindowAnimationTiming;
Callback FrameRequestCallback = void (DOMTimeStamp time);
```

C. WEBGL API

1. Cấu tạo của ứng dụng WebGL

Để hiển thị WebGL vào một trang, tối thiểu chúng ta phải thực hiện các bước sau:

- Tạo một đối tượng canvas.
- Tạo bối cảnh 3D.
- Khởi tạo viewport.
- Tạo một hoặc nhiều bộ đệm chứa dữ liệu hiển thị (thường là đỉnh).
- Tạo một hoặc nhiều ma trận để xác định phép chuyển đổi từ bộ đệm đỉnh sang màng hình.
- Tạo một hoặc nhiều shader để thực hiện thuật toán vẽ.
- Khởi tạo các shader với các tham số.
- Vẽ.

Hãy xem một vài ví dụ minh họa cho cấu trúc này.

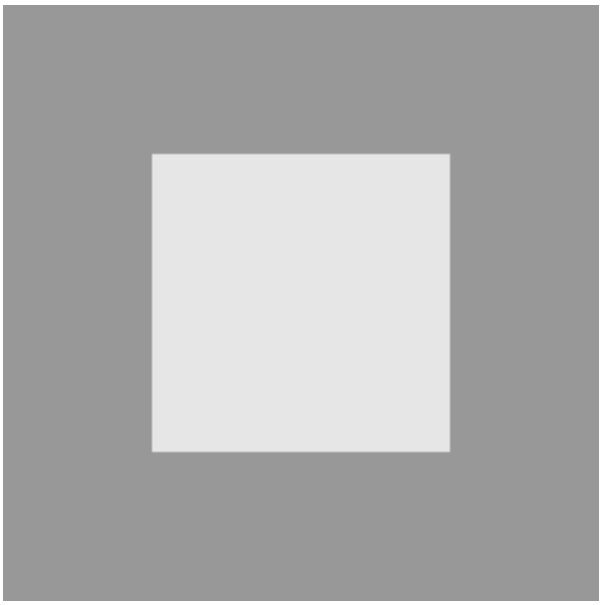
2. Ví dụ webgl đơn giản

Để minh họa cấu trúc cơ bản của WebGL, chúng ta sẽ viết rất đơn giản một hình vuông màu trắng duy nhất trên khung vẽ. Kết quả được thực hiện trong hình 2-1.

```
<!doctype html>
<html>
  <body>
    <canvas width = "570" height = "570" id = "my_Canvas"></canvas>
    <script>
      /*============ Creating a canvas =================*/
      var canvas = document.getElementById('my_Canvas');
      gl = canvas.getContext('experimental-webgl');
      /*========== Defining and storing the geometry =========*/
      var vertices = [
        -0.5,0.5,0.0,
        -0.5,-0.5,0.0,
        0.5,-0.5,0.0,
        0.5,0.5,0.0
      ];
      indices = [3,2,1,3,1,0];
      // Create an empty buffer object to store vertex buffer
      var vertex_buffer = gl.createBuffer();
      // Bind appropriate array buffer to it
      gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
      // Pass the vertex data to the buffer
      gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
      // Unbind the buffer
      gl.bindBuffer(gl.ARRAY_BUFFER, null);
      // Create an empty buffer object to store Index buffer
      var Index_Buffer = gl.createBuffer();
      // Bind appropriate array buffer to it
      gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, Index_Buffer);
```

```
// Pass the vertex data to the buffer
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new (indices),gl.STATIC_DRAW);
// Unbind the buffer
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
/*====================== Shaders =======================*/
// Vertex shader source code
var vertCode =
   'attribute vec3 coordinates;' +
   'void main(void) {' +
      ' gl_Position = vec4(coordinates, 1.0);' +
   '}';
// Create a vertex shader object
var vertShader = gl.createShader(gl.VERTEX_SHADER);
// Attach vertex shader source code
gl.shaderSource(vertShader, vertCode);
// Compile the vertex shader
gl.compileShader(vertShader);
// Fragment shader source code
var fragCode =
   'void main(void) {' +
      ' gl_FragColor = vec4(0.0, 0.0, 0.0, 0.1);' +
   '}';
// Create fragment shader object
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
// Attach fragment shader source code
gl.shaderSource(fragShader, fragCode);
// Compile the fragmentt shader
gl.compileShader(fragShader);
// Create a shader program object to
// store the combined shader program
var shaderProgram = gl.createProgram();
// Attach a vertex shader
gl.attachShader(shaderProgram, vertShader);
// Attach a fragment shader
gl.attachShader(shaderProgram, fragShader);
// Link both the programs
gl.linkProgram(shaderProgram);
// Use the combined shader program object
gl.useProgram(shaderProgram);
/* ======= Associating shaders to buffer objects =======*/
// Bind vertex buffer object
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
// Bind index buffer object
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, Index_Buffer);
// Get the attribute location
var coord = gl.getAttribLocation(shaderProgram, "coordinates");
// Point an attribute to the currently bound VBO
gl.vertexAttribPointer(coord, 3, gl.FLOAT, false, 0, 0);
// Enable the attribute
gl.enableVertexAttribArray(coord);
/*============= Drawing the Quad ================*/
// Clear the canvas
gl.clearColor(0.5, 0.5, 0.5, 0.9);
// Enable the depth test
gl.enable(gl.DEPTH_TEST);
// Clear the color buffer bit
```

```
        gl.clear(gl.COLOR_BUFFER_BIT);
        // Set the view port
        gl.viewport(0,0,canvas.width,canvas.height);
        // Draw the triangle
        gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT,0);
      </script>
    </body>
</html>
```



Hình 3-1. Hình vuông WebGL

3. Phần tử canvas và môi trường WebGL

Để đưa WebGL vào trang web của bạn, hãy tạo <canvas> gắn thẻ ở đâu đó trong trang, lấy đối tượng DOM được liên kết với nó (sử dụng document.getElementById()), và sau đó lấy bối cảnh WebGL cho nó.

Ví dụ 3-1. Cách lấy bối cảnh WebGL tử phần tử DOM canvas. Phương thức getcontext() có thể lấy một trong các chuỗi ngữ cảnh sau: “2d” cho 2D, “webgl” cho bối cảnh WebGL hoặc “experimental-webgl” cho các trình duyệt cũ hơn. Kiểu “exprimental-webgl” vẫn được hỗ trợ trong các trình duyệt mới, ngay cả khi vẫn hỗ trợ cổng “webgl”. Vì vậy chúng ta sẽ sử dụng “exprimental-webgl” để đảm bảo ngữ cảnh.

Ví dụ 3-1. Cài đặt bối cảnh WebGL từ đối tượng canvas

```
funtion initWebgl(canvas) {
        var gl = null;
        var msg = “Your browser does not support WebGL”;
        try
        {
                gl = canvas.getcContext(“exprimental-webgl”);
        }
        Catch (e)
        {
                msg = “Error” + e.toString();
        }
        if (!gl) {
```

```
            alert(msg);
            throw new Error(msg);
        }
        return gl;
}
```

4. Viewport

Khi đã có bối cảnh WebGL hợp lệ, cần cho nó biết giới hạn hình chữ nhật khung vẽ. Trong WebGL điều này được gọi là chế độ xem. Đặt chế độ xem trong WebGL chỉ cần gọi phương thức viewport(). Như trong ví dụ 3-2.

Ví dụ 3-2. Cài đặt chế độ xem

```
function initViewport(gl, canvas) {
        gl.viewport(0, 0, canvas.width, canvas.height);
}
```

Đối tượng gl được sử dụng xuyên suốt trong cả chương trình và được tạo duy nhất trong hàm initWebgl().

5. Bộ đệm, mảng bộ đệm, và loại mảng

Bộ đệm là vùng nhớ của GPU cấp cho WebGL để lưu trự dữ liệu (có các lại bộ đệm đỉnh, bộ đệm chỉ mục, bộ đệm khung,...)

Ví dụ 3-3. Cách tạo dữ liệu bộ đệm đỉnh cho hình vuông (1x1). Các đỉnh được lưu dưới dạng mảng JavaScript rồi truyền vào bộ đệm đỉnh.

Ví dụ 3-3. Tạo dữ liệu bộ đệm đỉnh

```
function createSquare(gl) {
        var vertexBuffer;
        vertexBuffer = gl.createBuffer();
        gl.bindBuffer(gl. ARRAY_BUFFER, vertexBuffer);
        var verts = [
                .5,     .5,     .0,
                -.5,    .5,     .0,
                .5,     -.5,    .0,
                -.5,    -.5,    .0
        ];
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
        var square = {buffer:vertexBuffer, vertSize:3, nVerts:4,
        primtype:gl.TRIANGLE_STRIP}
        return square;
}
```

6. Ma trận

Trước khi chúng ta có thể vẽ hình vuông, chúng ta phải tạo ra một vài ma trận để xác định vị trí của hình vuông trong hệ tọa độ 3D, liên quan đến camera. Đây được gọi là ma trân

modelView, vì nó kết hợp các biến đổi của mô hình (lưới 3D) và camera. Trong ví dụ phía dưới, chúng tôi đang chuyển đổi hình vuông bằng cách dịch nó theo trục z âm (di chuyển ra xa máy ảnh -3.333 đơn vị). Ma trận thứ 2 chúng ta cần lầm trận chiếu, sẽ được yêu cầu bởi trình đổ bóng để chuyển đội hệ tọa độ không gian 3D thành tạo độ 2D được vẽ lên màng hình. Trong ví dụ phía dưới, ma trận chiếu xác điịnh máy ảnh phối cảnh nhìn 45 độ.

Trong WebGL, ma trận được biểu diễn đơn giản dưới dạng mảng số; ví dụ, ma trận 4x4 có 16 phần tử kiểu Float32Array. Để thiết lập và thao tác, chúng tôi sử dụng một thư viện mã nguồn mở có tên là glMatrix.

Ví dụ

3-4. Cài đặt ma trận phép chiếu và modelView

```
var projectionMatrix, modelViewMatrix;
function initMatrices(canvas) {
        modelViewMatrix = mat4.create();
        mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -3.333]);

        projectionMatrix = mat4.create();
        mat4.perspective(projectionMatrix, Math.PI /4,canvas.width/canvas.height,
        1,10000);
}
```

7. Shader

Shader là các chương trình nhỏ được viết bằng GLSL (giống như C) xác định cách các pixel cho các đối tượng 3D thực sự được vẽ trên màng hình. WebGL yêu cầu nhà phát triển cung cấp trình đổ bóng cho từng đối tượng được vẽ. Trong thực tế, việc cung cấp một trình tạo bóng cho toàn bộ ứng dụng, sử dụng lại với các giá trị tham số và hình học khác nhau cho mỗi lần là đủ.

Một shader thường bao gồm 2 loại: vertex shader và fragment shader (còn được gọi là pixel shader).Vertex shader có nhiệm vụ chuyển đổi tọa độ của đối tượng thành không gian 2D; fragment shader có nhiệm tạo đầu ra là màu cuối cùng của từng pixel dựa vào đầu vào như màu sắc, texture, ánh sáng và giá trị vật liệu.

Trong WebGL, thiết lập shader yêu cầu một chuỗi các bước, bao gồm biên dịch các phần riêng lẻ từ mã nguồn GLSL, sau đó liên kết chúng lại với nhau. Ví dụ 3-5 đầu tiên chúng tôi xác định hàm trợ giúp, createShader(), sử dụng các phương pháp webgl để biên dịch các vertex và fragment shader từ mã nguồn.

Ví dụ 3-5. Mã nguồn Shader

```
function createShader(gl, str, type) {
        var shader;
```

```
        if (type == “fragment”) {
                shader = gl.createShader(gl.FRAGMENT_SHADER);
        } else if (type == “vertex”) {
                Shader = gl.createShader(gl.VERTEX_SHADER);
        } else return nulll;
        gl.shaderSource(shader, str);
        gl.compileShader(shader);
        if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
                alert(gl.getShaderInfoLog(shader));
                return null;
        }
        return shader;
}
```

Mã nguồn GLSL được cung cấp dưới dạng các chuỗi JavaScript

```
var vertexShaderSource =
        “attribute vec3 vertexPos;\n” +
        “uniform mat4 modelViewMatrix;\n” +
        “uniform mat4 projectionMatrix;\n” +
        “void main(void) {\n” +
        “       gl_Position = projectionMatrix * modelviewMatrix *\n” +
        “               vec4(vertexPos, 1.0);\n” +
        “}\n”;
var fragmentShaderSource =
        “void main(void) {\n” +
        “       gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n” +
        “}\n”;
```

Khi các thành phần của trình đổ bóng đã được biên dịch, chúng ta cần liên kết chúng lại với nhau thành một chương trình, sử dụng các phương thức gl.createProgram(), gl.attachShader() và gl.linkProgram(). Sayu khi liên kết thành công, chúng ta phải xừ lý từng biến được xác định trong mã shader GLSL để chúng có thể được khởi tạo với các giá trị từ mã JavaScript. Thực hiện điều này bằng các phương thức WebGL gl.getAttribLocation() và gl.getUniformLocation(). Được định nghĩa trong đoạn mã sau:

```
var shaderProgram, shaderVertexPositionAttribute,
        shaderProjectionMatrixUniform,
        shaderModelViewMatrixUniform;

function initShader(gl) {
        var fragmentShader = createShader(gl, fragmentShaderSource, “fragment”);
        var vertexShader = createShader(gl, vertexShaderSource, “vertex”);

        shaderProgram = gl.createProgram();
        gl.attachShader(shaderProgram, vertexShader);
        gl.attachShader(shaderProgram, fragmentShader);
        gl.linkProgram(shaderProgram);

        shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram,
        “vertexPos”);
        gl.enableVertexAttribArray(shaderVertexPositionAttribute);
```

```
        shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram,
        “projectionMatrix”);

        shaderModelViewMatrixUniform = gl.getUniformLocation(shader,
        “modelViewMatrix”);
        if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
                alert(“Could not initalise shader”);
        }
}
```

8. Vẽ

Đầu tiên, draw() xóa khung vẽ với màu nền đen. Phương thức gl.clearColor(). Phương thức này có thành phần RGBA (đỏ, xanh lá, xanh dương, alpha). Lưu ý rằng các giá trị WebGL’s RGBA là các số có dấu phẩy động từ 0.0 đến 1.0 (trái ngược với phạm vi số nguyên 0 đến 255 được sử dụng cho các giá trị màu web như trong CSS). Sau đó, gl.clear() xóa bộ đệm màu WebGl, đó là khu vực trong bộ nhớ GPU được sử dụng để hiển thị các bit trên màng hình.

Tiếp theo, hàm draw() thiết lập (liên kết) bộ đjêm đỉnh cho hình vuông được vẽ, thiết lập (sử dụng) shader sẽ được thực thi để vẽ, và kết nối bộ đệm đỉnh và ma trận với shader làm đầu vào.

Cuối cùng, chúng ta gọi phương thức drawArrays() để vẽ.

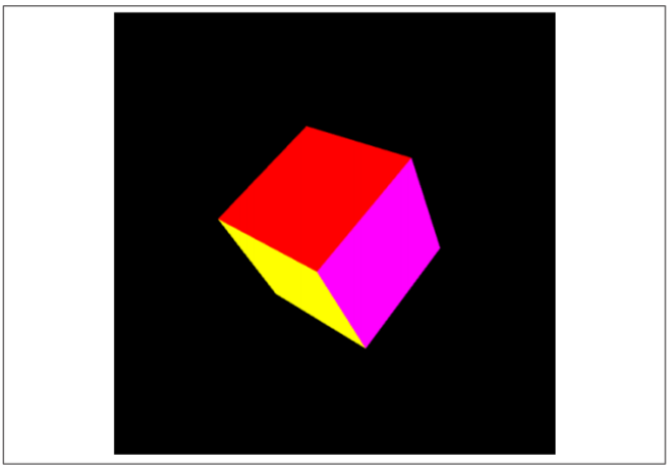
Ví dụ 3-6. Mã nguồn vẽ

```
function draw(gl) {
        gl.clearColor(0.0, 0.0, 0.0, 1.0);
        gl.clear(gl.COLOR_BUFFER_BIT);

        gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
        gl.useProgram(shaderProgram);
        vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false,
        0, 0);
        gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
        gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);
        gl.drawArrays(obj.primtype, 0, obj.nVerts);
}
```

9. Tạo hình học 3D

Bây giờ chúng ta sẽ sử dụng WebGL để thực hiện bản vẽ khối lập phương với nhiều màu sắc. Và chúng ta sẽ cho nó vài chuyện động nhỏ để chúng ta có thể nhìn thấy nó từ mọi phía. Hình 3-2 cho thấy một ảnh chụp màng hình của khối lập phương.


Hình 3-2. Khối lập phương

Ví dụ 3-7. Mã nguồn cài đặt khối lập phương

```
function createCube(gl) {
    var vertexBuffer;
    verttexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
        //front face
        -1.0, -1.0, 1.0,
        1.0, -1.0, 1.0,
        1.0, 1.0, 1.0,
        -1.0, 1.0, 1.0,
        //back face
        -1.0, -1.0, -1.0,
        -1.0, 1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, -1.0, -1.0,
        //top face
        -1.0, 1.0, -1.0,
        -1.0, 1.0, 1.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, -1.0,
        //bottom face
        -1.0, -1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0,
        -1.0, -1.0, 1.0,
        //right face
        1.0, -1.0, -1.0,
        1.0, 1.0, -1.0,
        1.0, 1.0, 1.0,
        1.0, -1.0, 1.0,
        //left face
        -1.0, -1.0, -1.0,
        -1.0, -1.0, 1.0,
        -1.0, 1.0, 1.0,
        -1.0, 1.0, -1.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
```

Tiếp theo, chúng tôi tạo dữ liệu màu, một màu bốn thành phần cho mỗi đỉnh và lưu nó trong colorBufer.

```
var colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
var faceColors = [
        [1.0, 0.0, 0.0, 1.0], //front face
        [0.0, 1.0, 0.0, 1.0], //back face
        [0.0, 0.0, 1.0, 1.0], //top face
        [1.0, 1.0, 0.0, 1.0], //bottom face
        [1.0, 0.0, 1.0, 1.0], //right face
        [0.0, 1.0, 1.0, 1.0]  //left face
];
var vertexColors = [];
for (var i in faceColors) {
        var color = faceColors[i];
        for (var j = 0, j < 4, j++)
                vertexColors = vertexColors.concat(color);
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors), gl.STATIC_DRAW);
```

Cuối cùng, chúng ta sẽ tạo một bộ đệm chỉ mục, để giữ một tập hợp các chỉ mục vào dữ liệu bộ đệm đỉnh. Chúng tôi luu trữ điều này trong biến cubeIndexBuffer. Bộ đệm được lập chỉ mục cho phép dữ liệu được lưu trữ gọn hơn bằng cách tránh lặp lại dữ liệu.

```
var cubeIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer);
var cubeIndices = [
        0, 1, 2,        0, 2, 3,        //front face
        4, 5, 6,        4, 6, 7,        //back face
        8, 9, 10,       8, 10, 11,      //top face
        12, 13, 14.     12, 14, 15,     //bottom face
        16, 17, 18,     16, 18, 19,     //right face
        20, 21, 22,     20, 22, 23      //left face
];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Unit16Array(cubeIndices),
gl.STATIC_DRAW);
var cube = {buffer:vertexBuffer, colorBufffer:colorBuffer, indices:cubeIndexBuffer,
vertSize:3, nVerts:24, colorSize:4, nColors:24, nIndices:36, primtype:gl.TRIANGLES};
return cube;
}
```

Để các khối màu được vẽ, chúng ta phải được chuyển đến bộ đổ bóng.

```
var vertexShaderSource =
“       attribute vec3 vertexPos;\n” +
“       attribute vec4 vertexColor;\n” +
“       uniform mat4 modelViewMatrix;\n” +
“       uniform mat4 projectionMatrix;\n” +
“       varying vec4 vColor;\n” +
“       void main(void) {\n” +
“       gl_Position = projectionMatrix * modelViewMatrix * vec4(vertexPos, 1.0);\n” +
“               vColor = vertexColor;\n” +
“       }\n”;
```

```
Var fragmentShaderSource =
“	precision mediump float;\n” +
“	varying vec4 vColor;\n” +
“	void main(void) {\n” +
“		gl_FragColor = vColor;\n” +
“	]\n”
```

Tiếp theo, chúng ta phải liên kết các bộ đệm màu và chỉ mục được tạo trước đó trong hàm createCube(). Cuối cùng dùng phương thức gl.drawElements() thay vì gl.drawArray().

```
function draw(gl, obj) {
	gl.clearColor(0.0, 0.0, 0.0, 1.0);
	gl.enableColor(gl.DEPTH_TEST);
	gl.clear(gl.COLOR_BUFFER_BIT | glDEPTH_BUFFER_BIT);

	gl.useProgram(shaderProgram);

	gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
	gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT,
	false, 0, 0);
	gl.bindBuffer(gl.ARRAY_BUFFER, obj.colorBuffer);
	gl.vertexAttribPointer(shaderVertexColorAttribute, obj.colorSize, gl.FLOAT, false,
	0, 0);
	gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);
	gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
	gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);

	gl.drawElements(obj.primtype, obj.nIndices, gl.UNSIGNED_SHORT, 0);
}
```

10. Thêm hiệu ứng động

Bây giờ chúng ta sẽ sử dụng một kỹ thuật hoạt hình rất đơn giản để làm khối lập phương xoay xung quanh một trục. Ví dụ 2-8. Hàm animate() xoay khối lập phương xung quanh phép quay được các định trước đó trong một khoảng thời gian vài giây.

Ví dụ 2-8. Hiệu ứng động

```
var duration = 5000;		//ms
var currentTime = Date.now();

function animate() {
	var now = Date.now();
	var delat = now – currentTime;
	currentTime = now;
	var fract = delat /duration;
	var angle = Math.PI * 2 *fract;
	mat4.rotate(modelViewMatrix, modelViewMatrix, angle, rotationAxis);
}

function run(gl, cube) {
	requestAnimationFrame(function() { run(gl, cube);});
```

```
        draw(gl, cube);
        animate();
}
```

11. Sử dụng Texture

Texture là hình ảnh bitmap được hiển thị trên bề mặt hình học. Tạo dữ liệu hình ảnh cho texture dùng đối tượng hình ảnh DOM, nghĩa là bạn có thể cugn cấp các định dạng hình ảnh web tiêu chuẩn, chẳng hạn như JPEG và PNG, cho WebGL dưới dạng texture bằng cách chỉ cần đặt thuộc tính Image src.

Ví dụ 3-9. Texture

```
var okToRun = false;

function handleTextureLoaded(gl, texture) {
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
        gl.textureImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
        gl.UNSIGNED_BYTE, texture.iamge);
        gl.texParameteri(gl.TEXTURE_2D, gl.TECTURE_MAG_FILTER, gl.NEAREST);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
        okToRun = true;
}

var webGLTexture;
function initTexture(gl) {
        webGLTexture = gl.createTexture();
        webGLTexture.image = new Image();
        webGLTexture.image.onload = function() {
                handleTextureLoaded(gl, webGLTexture)
        }
        webGLTexture.image.src = "../images/image.jpg";
}
```

Giờ chúng ta sẽ bắt đầu code thử khối lập phương được dán texture. Do mọi mặt đều được dán một texture nên bất ký góc nào của mặt khối đều nằm ở 1 góc của họa tiết, ví dụ: [0, 0], [0, 1], [1, 0], [1, 1]. Lưu ý rằng thứ tự các giá trị này phải tương ứng với thứ tự của các đỉnh trong bộ đệm đỉnh. Ví dụ 3-10 biểu thị mã để tạo bộ đệm tọa độ texture.

Ví dụ 3-10. Mã tạo bộ đệm cho khối lập phương dán texture

```
var texCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);
var textureCoords = [
        //front face
        0.0, 0.0,
        1.0, 0.0,
        1.0, 1.0,
        0.0, 1.0,
        //back face
```

```
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            //top face
            0.0, 1.0,
            0.0, 0.0,
            1.0, 0.0,
            1.0, 1.0,
            //bottom face
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            1.0, 1.0,
            //right face
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0,
            0.0, 0.0,
            //left face
            0.0, 0.0,
            1.0, 0.0,
            1.0, 1.0,
            0.0, 1.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
gl.STATIC_DRAW);
```

Chúng ta phải sửa đổi mã shader để sử dụng thông tin texture thay vì màu. Trình tạp bóng xác định một thuộc tính đỉnh texCoord được truyền với dữ liệu đỉnh và một đầu ra khác nhau, vTexCoord, sẽ được gửi đến trình tạo bóng cho mỗi đỉnh. Sau đó, fragment shader sử dụng tọa độ texture này như là một chỉ mục vào dữ liệu texture map, được truyền dưới dạng thống nhất cho fragment shader trong biến uSampler. Mã được cập nhập trong ví dụ 2-10.

Ví dụ 3-11. Mã shader cho khối lập phương dán texture

```
var vertexShaderSource =
        “       attribute vec3 vertexPos;\n” +
        “       attribute vec2 texCoord;\n” +
        “       uniform mat4 modelViewMatrix;\n” +
        “       uniform mat4 projectionMatrix;\n” +
        “       varying vec2 vTexCoord;\n” +
        “       void main(void) {\n” +
                    gl_Position = projectionMatrix * modelViewMatrix *
                            vec4(vertexPos, 1.0);\n” +
        “           vTexCoord = texCoord;\n” +
        “       }\n”;
var fragmentShaderSource =
        “       precision mediump float;\n” +
        “       varying vec2 vTexCoord;\n” +
        “       uniform sampler2D uSampler;\n” +
        “       void main(void) {\n” +
        “           gl_FragColor = texture2D(uSampler, vec2(vTexCoord.s,
                    vTexCoord.t));\n” +
```

```
“        }\n”;
```

Ví dụ 3-12. Cài đặt dữ liệu texture để vẽ

```
gl.vertexAttribPointer(shaderTexCoordAttribute, obj.texCoordSize, gl.FLOAT, fasle, 0, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);

gl.uniformMatrix4fv(shaderProjectionMatrixUniform, fasle, projectionMatrix);
gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);

gl.activeTexture(gl.TEXTURE);
gl.bindTexture(gl.TEXTURE_2D, webGlTexture);
gl.uniformi(shaderSampleUniform, 0);
```

Hình 2-3. Texture3D

D. THREE.JS CÔNG CỤ 3D JAVASCRIPT

1. Dự án hàng đầu

Có lẽ dự án WebGL nổi tiếng nhất cho đến nay là RO.ME "3 Dreams of Black", một tác phẩm tương tác được tạo ra vào năm 2011 bởi nhà làm phim Chris Milk với sự giúp đỡ từ các kỹ sư của Google. Bộ phim là người bạn đồng hành với bài hát Black Black của ROME, một dự án âm nhạc của Danger Mouse và Daniele Luppi, với sự tham gia của Jack White và Norah Jones. Xem hình 3-1.

Figure 3-1. RO.ME "3 Dreams of Black," an interactive video experience inspired by the song "Black" from the album ROME

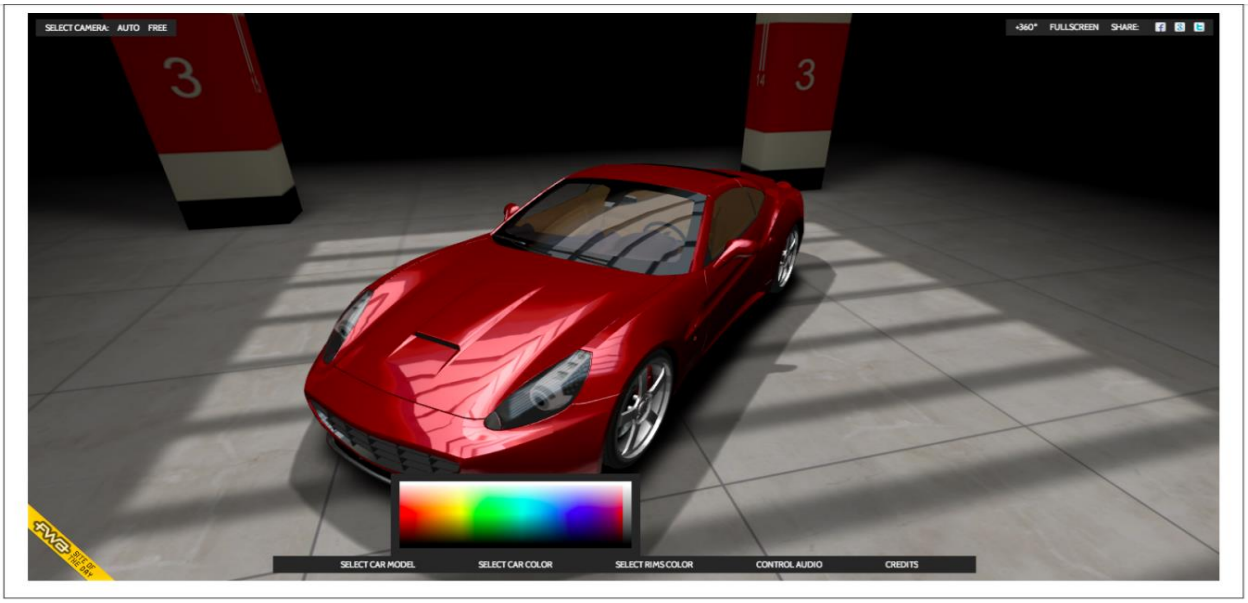


Figure 3-2. A car configurator and visualizer, by Plus 360 Degrees

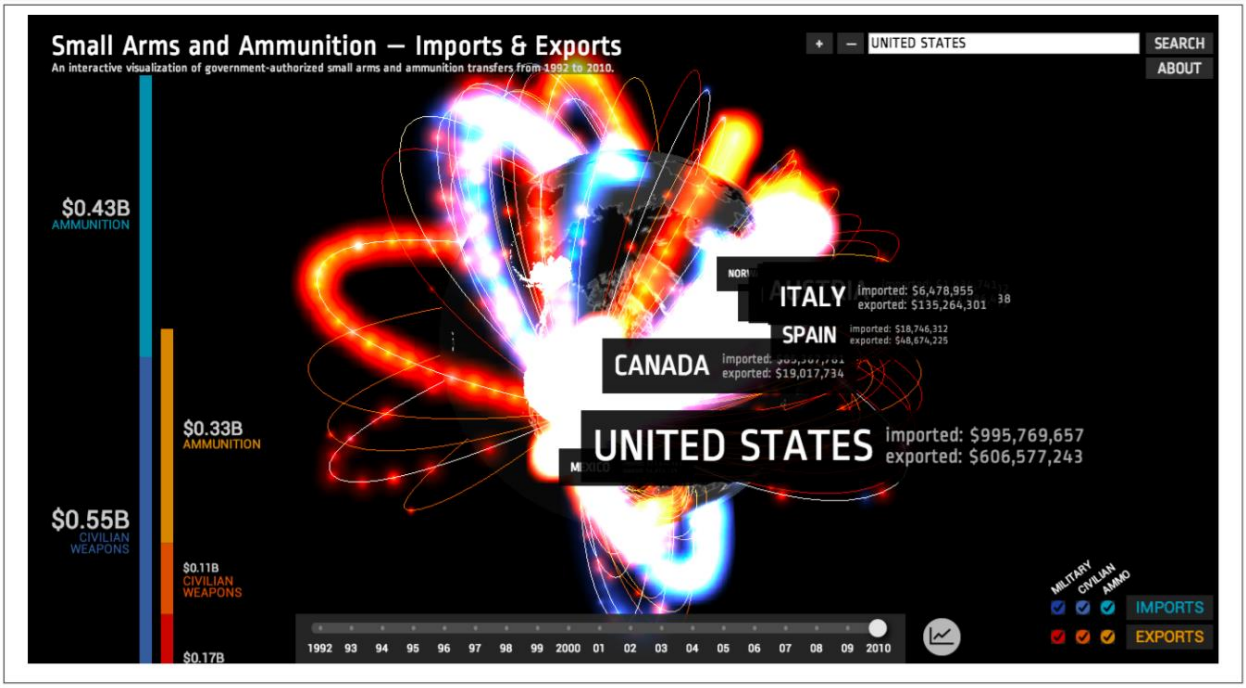


Figure 3-3. Small Arms Imports/Exports, a Google Experiment created by Google Ideas

2. Cài đặt Three.js

Để phát triển với Three.js, trước tiên bạn cần phải nhận gói mới nhất từ GitHub. https://github.com/mrdoob/three.js

3. Ví dụ đơn giản

Ví dụ 4-1. Tạo khối lập phương dán texture với Three.js

```
<sctipt type = “text/javascript”>
        var renderer = null;
        scene = null;
        camera = null;
        cube = null;

        var duration = 5000;
        var currentTime = Date.now();

        function animate() {
                var now = Date.now();
                var deltat = now – curentTime;
                currentTime = now;
                var fract = deltat / duration;
                var angle = Math.PI * 2 *fract;
                cube.rotation.y += angle;
        }

        function run() {
                requestAnimationFrame(function() { run();});
```

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

        $(document).ready(
                function() {
                        var canvas = document.getElementById("webglcanvas");

                        renderer = new THREE.WebGLRenderer({canvas:canvas,
                        antialias: true});

                        renderer.setSize(canvas.width, canvas.height);
                        scene = new THREE.Scene();

                        camera = new THREE.PerpectiveCamera(45, canvas.width /
                        canvas.height, 1, 4000);
                        scene .add(camera);

                        var mapUrl = "../images/image.jpg";
                        var map = THREE.ImageUtils.loadTexture(mapUrl);
                        var material = new THREE.MeshBasicMaterial({map:map});

                        var geometry = new THREE.CubeGeometry(2, 2, 2);
                        cube = new THREE.Mesh(geometry, material);

                        cube.position.z = -8;
                        cube.rotation.x = Math.PI / 5;
                        cube.rotation.y = Math.PI / 5;
                        scene.add(cube);

                        run();
                }
        );
</script>
```

4. Tạo Renderer

Đầu tiên, chúng ta cần tạo renderer. Three.js sử dụng hệ thống renderer plug-in. Ví dụ, chúng ta có thể kết xuất cảnh tương tự bằng các API khác nhau, như WebGL hoặc API canvas 2D.

```
// Tạo three.js renderer và liên kết với canvas
renderer = new THREE.WebGLRenderer({canvas:canvas, antialias: true});

// Cài đặt kích thước khung nhìn
renderer.setSize(canvas.width, canvas.height);
```

5. Tạo khung cảnh

Tiếp theo, chúng tôi tạo một cảnh bằng cách tạo một đối tượng THREE.Scene mới. Cảnh là đối tượng cấp cao nhất trong hệ thống phân cấp đồ họa Three.js. Nó chứa tất cả các đối tượng đồ họa khác.

Khi chúng ta đã có Scene, cần phải thêm một vài đối tượng vào: camera và mesh. Camera xác định vị trí đặt mắt xem. THREE.PerspectiveCamera, khởi tạo với góc nhìn 45 độ, kích thước khung nhìn, giá trị mặt cắt trước và sau.

Mã để tạo cảnh và thêm camera:

```
// Tạo three.js scene
scene = new THREE.Scene();

camera = new THREE.PerspectiveCamera(45, canvas.width / canvas.height, 1, 4000);
scene.add(camera);
```

Và bây giờ là thêm lưới vào cảnh.

```
// Đầu tiên tạo texture
var mapUrl = "../images/image.jpg";
var map = THREE.ImageUtils.loadTexture(mapUrl);

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

Cuối cùng, chúng ta tạo cube mesh.

```
cube.position.z = -8;
cube.rotation.x = Math.PI / 5;
cube.rotation.y = Math.PI / 5;

scene.add(cube);
```

6. Thực hiện vòng lặp

Như với ví dụ phần trước, chúng ta phải thực hiện một vòng lặp chạy bằng cách sử dụng requestAimationFrame(). Nhưng sừ dụng Three.js, chúng ta chỉ cần gọi:

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

Và thư viện làm phần còn lại.

```
var duration = 5000;
var currentTime = Date.now();
function animate() {
        var now = Date.now();
        var deltat = now – currentTime;
        currentTime = Date.now();
        var fract = deltat / duration;
        var angle = Math.PI * 2 * fract;
        cube.rotation.y += angle;
}

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

7. Light

Thêm hướng chiếu ánh sáng, ghóp phần làm hiệu ứng 3D thêm thật hơn.

Ví dụ 3-2. Hướng ánh sáng vào khối lập phương

```
var light = new THREE.DirectionalLight(0xffffff, 1.5);
light.position.set(0, 0, 1);
scene.add(light);
var mapUrl = "../images/image.jpg";
var map = THREE.ImageUtils.loadTexture(mapUrl);

var material = new THREE.MeshPhongMaterial({ map: map});
```

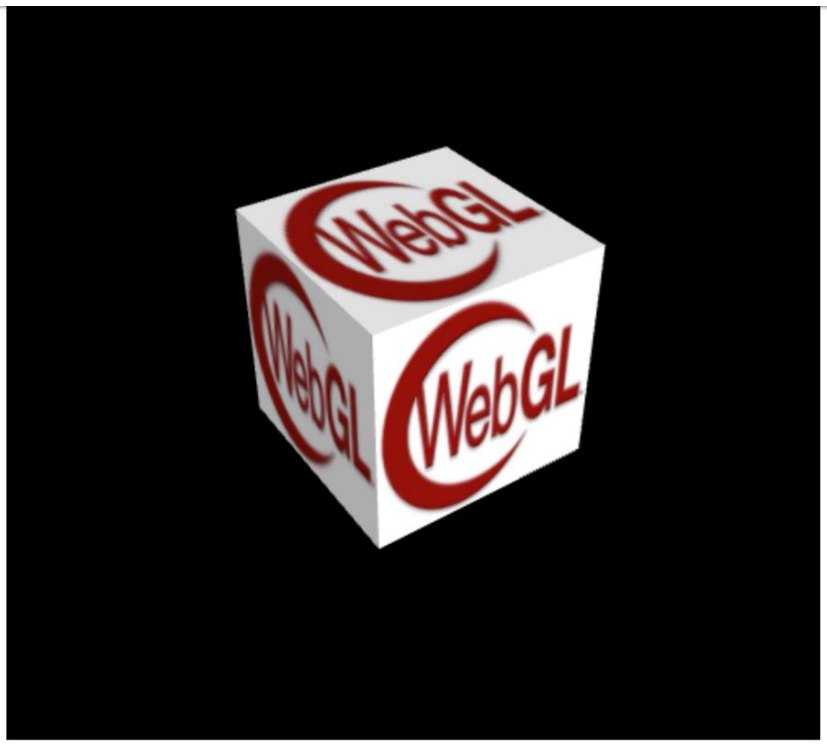


Figure 3-6. Three.js cube with lighting and Phong shading

E. ỨNG DỤNG "3D HOUSE"

1. Giới thiệu

Ứng dụng được thực hiện dựa trên nền tảng HTML5 và WebGL, mã nguồn hỗ trợ thực hiện là three.js.

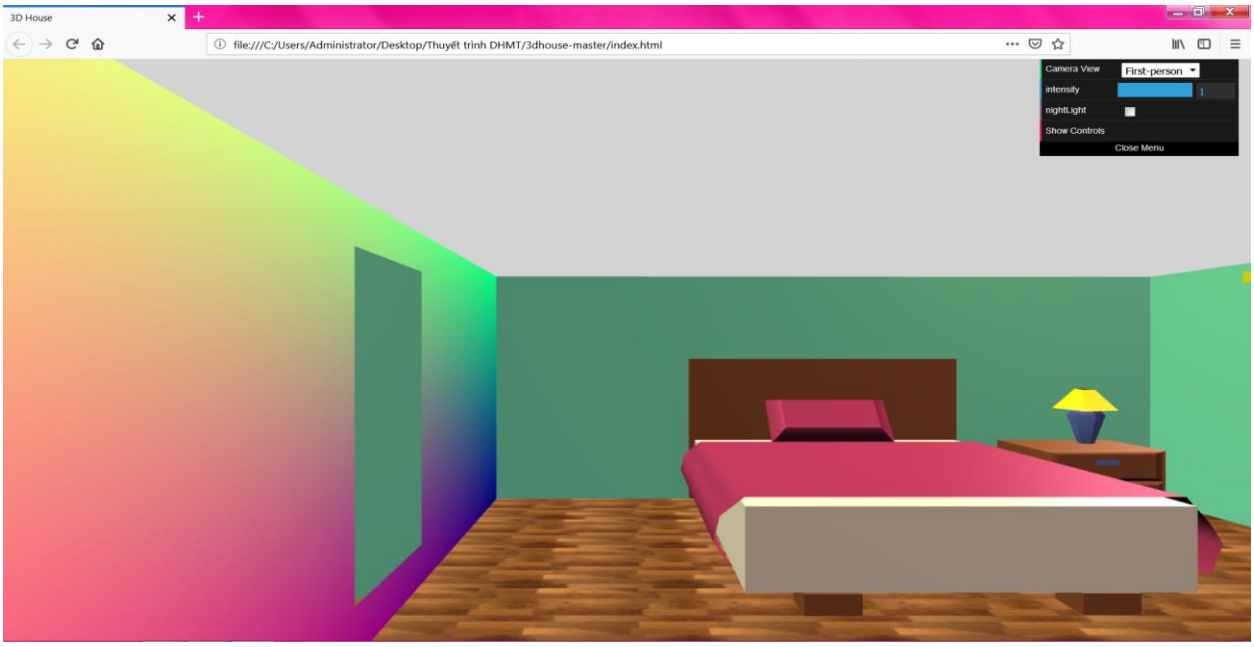
Ứng dụng mô phỏng mô hình 3D của một ngôi nhà cố định với các vị trí, thiết bị màu sắc cố định sẵn.

Ứng dụng cho phép xoay, thay đổi góc nhìn, thay đổi độ sáng, bật đèn, thay đổi nền nhà,..

Để ứng dụng chạy được hoàn chỉnh, bận nên mở nó bởi trình duyệt firefox.

Góc nhìn top-down

Góc nhìn ngang tại phòng ngủ (bedroom)



2. Cấu trúc chương trình

- Libraries:
  - dat.gui.js
  - dat.gui.min.js
  - FirtPeronControls.js
  - LoadingManager.js
  - MTLLoader.js
  - OBJLoader.js
  - OBJMTLoader.js
  - OrbitControl.js
  - Stats.min.js
  - three.js
  - three.min.js

  Chứa các tệp js cần thiết để thực hiện.

- Model:
  - Convert_obj_three.js

  Chuyển dổi đối tượng trong three.js

- House.js
  Tệp thực thi chính sẽ gọi các file trong libaries và model

- Texture: chứa các file hình ảnh
- Index.html: thực thi file house.js

3. Phân tích chương trình

Vùng biến kiểm soát

```
// Global variables
var RENDER_WIDTH = window.innerWidth, RENDER_HEIGHT = window.innerHeight;
var controls,renderer,scene,camera, container;
var ground,houseContainer;
var groundMaterial,texture_update;
var light;
var gui;
var mouse = { x: 0, y: 0 }, INTERSECTED;
var targetList = [];               // list of objects to highlight
var collisionList = [];            // list of objects to collide with
var clock = new THREE.Clock();     // clock used for firstPerson controls
var canvas1, context1, textureC;
var firstPerson = false;           // toggle to see if in firstPerson view
var guiDestroyFlag = false;        // toggle used to indicate whether the current GUI needs to be destroyed
var roofMaterial;
// floor planes
var plane, kitchenPlane, bedPlane, bathroomPlane, diningPlane, livingPlane;
// floor textures
var textureBed, textureBath, textureKitchen, textureLiving, textureDining, textureHall1, textureHall2, textureGround;
// for mirror material
var bathMirCube, bathMirCubeCamera;
var bedMirCube, bedMirCubeCamera;
// lights
var bedroomLamp, diningroomLight, livingroomLight, bathroomLight, kitchenLight;
```

Hàm khởi tạo:

Khởi tạo môi trường

```
container = document.getElementById("container");

renderer = new THREE.WebGLRenderer({ antialias:true });
renderer.setSize(RENDER_WIDTH,RENDER_HEIGHT);
renderer.domElement.style.backgroundColor = '#000000';
renderer.shadowMapEnabled = true;
document.body.appendChild( renderer.domElement );

container.appendChild(renderer.domElement);
scene = new THREE.Scene();

// add camera to the scene
camera = new THREE.PerspectiveCamera(45, RENDER_WIDTH / RENDER_HEIGHT, 0.1, 10000);
camera.position.y = 240;
camera.position.x = 20;
camera.position.z = -20;
scene.add(camera);
```

Dán texture

```
// Initialize floor textures
textureBed = THREE.ImageUtils.loadTexture( "texture/floor3.jpg" );
setFloorTextureProperties(textureBed);
textureBath = THREE.ImageUtils.loadTexture( "texture/floor2.jpg" );
setFloorTextureProperties(textureBath);
textureKitchen = THREE.ImageUtils.loadTexture( "texture/floor1.jpg" );
setFloorTextureProperties(textureKitchen);
textureLiving = THREE.ImageUtils.loadTexture( "texture/floor.jpg" );
setFloorTextureProperties(textureLiving);
textureDining = THREE.ImageUtils.loadTexture( "texture/floor4.jpg" );
setFloorTextureProperties(textureDining);
textureHall1 = THREE.ImageUtils.loadTexture( "texture/floor2.jpg" );
setFloorTextureProperties(textureHall1);
textureHall2 = THREE.ImageUtils.loadTexture( "texture/floor1.jpg" );
setFloorTextureProperties(textureHall2);
textureGround = THREE.ImageUtils.loadTexture( "texture/grass.png" );
setGroundTextureProperties(textureGround);
```

Mái nhà và mặt đất

```
// Ground plane
var planeGeometry = new THREE.PlaneBufferGeometry( 300, 420 );
var planeMaterial = new THREE.MeshLambertMaterial( {map: textureGround, side: THREE.DoubleSide} );
plane = new THREE.Mesh( planeGeometry, planeMaterial );
plane.position.set(50, 0, -30);
plane.receiveShadow = true;
plane.castShadow = false;
plane.rotation.x = 1.57;
scene.add( plane );

// Roof plane
var roofPts = [];
roofPts.push( new THREE.Vector2 ( 0, 0 ) );
roofPts.push( new THREE.Vector2 ( 151, 0 ) );
roofPts.push( new THREE.Vector2 ( 151, 189 ) );
roofPts.push( new THREE.Vector2 ( 117, 189 ) );
roofPts.push( new THREE.Vector2 ( 117, 164 ) );
roofPts.push( new THREE.Vector2 ( 0, 164 ) );
roofPts.push( new THREE.Vector2 ( 0, 0 ) );

var roofShape = new THREE.Shape( roofPts );
var roofGeometry = new THREE.ShapeGeometry( roofShape );
roofMaterial = new THREE.MeshBasicMaterial( {color: 0xd3d3d3, side:
THREE.DoubleSide, transparent: true} );
var roof = new THREE.Mesh( roofGeometry, roofMaterial );
roof.position.set( -44,25.2,-134 );
roof.rotation.x = 1.57;
roofMaterial.opacity = 0;
scene.add( roof );
```

Phòng khách

```
// Living room floor plane
var livingPts = [];
livingPts.push( new THREE.Vector2 ( 0, 0 ) );
livingPts.push( new THREE.Vector2 ( 60.6, 0 ) );
livingPts.push( new THREE.Vector2 ( 60.6, 122 ) );
livingPts.push( new THREE.Vector2 ( 27, 122 ) );
livingPts.push( new THREE.Vector2 ( 27, 96 ) );
livingPts.push( new THREE.Vector2 ( 0, 96 ) );
livingPts.push( new THREE.Vector2 ( 0, 0 ) );

var livingShape = new THREE.Shape( livingPts );
var livingGeometry = new THREE.ShapeGeometry( livingShape );
var livingMaterial = new THREE.MeshLambertMaterial( {map: textureLiving, side:
THREE.DoubleSide} );
livingPlane = new THREE.Mesh( livingGeometry, livingMaterial );
// Set position and rotate to be flat against ground
livingPlane.position.set( 47,.4,-66.7 );
livingPlane.rotation.x = 1.57;
livingPlane.name = "Living Room";
scene.add( livingPlane );
// allow mesh to be clicked
targetList.push(livingPlane);
```

Thay đổi giao diện người dùng top-down:

function makegui1() {

...

}

- Thay đổi các giá trị giao diện
- Tạo khung giao diện tương tác
- Thay đổi texture nền nhà cho các phòng

Thay đổi giao diện người dùng first-person:

fucntion makeGui2() {

...

}

- Thay đổi các giá trị dao diện
- Tạo khung giao diện tương tác mới
- Tùy theo vị trí phòng mà quang cảnh xuất hiện khác nhau (ví dụ bedroom sẽ khác livingroom)

Tài liệu tham khảo

1. HTML5 tutorial (https://html5tutorial.net/)
2. WebGl tutorial (https://www.tutorialspoint.com/webgl/index.htm/)
3. Tony Parisi. Programming 3D Applications with HTML5 and WebGL, 13/02/2014.
4. Các ứng dụng 3D nền tảng WebGL trên Github (https://github.com/)