2012

YIN YANG

HTML5 Canvas - Lập Trình Game 2D

v1.0

Lý thuyết và demo thực hành về lập trình game 2D với API Canvas trong Html5

http://vietgamedev.net/
http://yinyangit.wordpress.com/
1/7/2012

LỜI TỰA

Flash là một công nghệ rất hiệu quả, phổ biến và cho phép lập trình viên có thể tạo ra những ứng dụng với đầy đủ các hiệu ứng hình ảnh, âm thanh đặc sắc. Những công nghệ tương tự như Java Applet hay một "đứa con" sáng giá của Microsoft là Silverlight cũng không thể đứng vững và cạnh tranh được với Flash. Nhưng một vấn đề nảy sinh ở đây là khả năng tương tác giữa các công nghệ này với các thành phần xung quanh nó (như các thẻ HTML) dường như không thể. Chúng bị cô lập và hoạt động độc lập với thế giới bên ngoài.

Giải pháp là quay trở lại sử dụng thuần HTML, Javascript và CSS, lập trình viên vẫn có thể tạo được ra ứng dụng với hiệu ứng đặc biệt và không bị các giới hạn mà những công nghệ trên gặp phải. Nhưng trở ngại lớn nhất là không có đủ API để tạo ra được những ứng dụng tương tự như trên Flash. Và tốc độ của các ứng dụng thuần HTML khá chậm, hầu như không thể chấp nhận được với một game có yêu cầu cấu hình trung bình.

Nhưng với sự ra đời của HTML5 cùng với các thành phần và API mới, giới hạn trên đã bị phá bỏ và đang từng bước thay thế dần các công nghệ như Flash. Với các ứng dụng cần những hiệu ứng đồ họa và chuyển động đặc biệt, lập trình viên có thể sử dụng Canvas với kiểu bitmap hoặc SVG với kiểu vector. Không chỉ áp dụng cho việc thiết kế các trang web trực quan, HTML5 còn được áp dụng để tạo ra các thư viện đồ họa giúp tạo ra các ứng dụng đồ thị, game trong cả môi trường 2D và 3D như những ứng dụng trên desktop.

Một điều đáng mừng nữa là HTML, Javascript và CSS không còn bị giới hạn trên trình duyệt mà có thể được triển khai trên desktop dưới dạng các widget, trên các thiết bị di động và có thể bất kì thiết bị nào. Như vậy, lập trình viên không cần sử dụng hay yêu cầu người dùng cài đặt bất kì thư viện nào để có thể chạy được các ứng dụng của họ. Một lợi thế rất lớn mà chỉ có HTML mới có thể đạt được. Tuy nhiên việc xây dựng game trên trình duyệt có thể là một trải nghiệm khó khăn vì phải cân nhắc giữa việc chọn lựa giữa các thư viện hiện đại, đầy đủ chức năng hay làm theo các API cấp thấp của HTML (thông qua Javascript).

Quá trình thực hiện sách này không thể tránh khỏi sai sót, bạn đọc có thể gửi phản hồi tại http://vietgamedev.vn hoặc blog http://yinyangit.wordpress.com hoặc gửi email trực tiếp cho tôi (yinyang.it@gmail.com) để thắc mắc, trao đổi cũng như giúp tôi sửa đổi, cập nhật nếu cần thiết.

Xin cảm ơn!

Mục lục

A. GIỚI THIỆU

HTML5 được hỗ trợ hầu trên tất cả trình duyệt. Nó là một tập hợp các tính năng đặc biệt. nhưng ta có thể tìm thấy hỗ trợ cho một số phần đặc trưng như canvas, video hoặc định vị địa lý. Những đặc điểm kỹ thuật HTML5 cũng xác định làm thế nào những dấu ngoặc nhọn tương tác với JavaScrip, thông qua các tài liệu thông qua các tài liệu Object Model (DOM) HTML5 không chỉ xác định một tag `<video>`, đó cũng là một DOM API tương ứng cho các đối tượng video trong DOM. Bạn có thể sử dụng API này để tìm kiếm hỗ trợ cho các định dạng video khác nhau, nghe nhạc, tạm dừng một đoạn video, mute audio , theo dõi bao nhiêu video đã được tải về, và mọi thứ khác bạn cần phải xây dựng một trải nghiệm người dùng phong phú xung quanh tag `<video>` .

- **Không cần phải vứt bỏ bất kì thứ gì:**

Ta không thể phủ nhận rằng HTML 4 là các định dạng đánh dấu thành công nhất từ trước đến nay. HTML5 được xây dựng dựa trên thành công đó. Bạn không cần phải bỏ những đánh dấu hiện có. Bạn không cần phải học lại những điều bạn đã biết. Nếu ứng dụng web của bạn trước đây sử dụng HTML 4, nó vẫn sẽ hoạt động trong HTML5.
Bây giờ, nếu bạn muốn cải thiện các ứng dụng web, bạn đã đến đúng nơi. Ví dụ cụ thể: HTML5 hỗ trợ tất cả các hình thức kiểm soát từ HTML 4, 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, những thành phần khác tinh tế hơn.
Ví dụ, các loại email input trông giống như một textbox, nhưng các trình duyệt linh động sẽ tùy biến bàn phím trên màn hình của họ để có thể dễ dàng hơn khi nhập địa chỉ email. Các trình duyệt cũ không hỗ trợ các loại email input sẽ xem nó như là một văn bản thông thường, và hình thức vẫn làm việc không có thay đổi đánh dấu hoặc kịch bản hack. Điều này có nghĩa là bạn có thể bắt đầu cải thiện các hình thức web của bạn ngày hôm nay, ngay cả khi một số khách truy cập vào web của bạn.

- **Rất dễ dàng để bắt đầu:**

"Nâng cấp" lên HTML5 có thể đơn giản chỉ bằng việc thay đổi thẻ DOCTYPE của bạn. DOCTYPE cần phải nằm trong dòng đầu tiên của tất cả các trang HTML. Các phiên bản trước của HTML được định nghĩa rất nhiều loại doctype, và lựa chọn một doctype đúng rất rắc rối. Trong HTML5 chỉ có một DOCTYPE:

```
<!DOCTYPE html>
```

Nâng cấp lên DOCTYPE HTML5 sẽ không phá vỡ cấu trúc tài liệu của bạn, bởi vì các thẻ lỗi thời trước đây được định nghĩa trong HTML 4 vẫn được vẽ ra trong HTML5. Nhưng nó cho phép bạn sử dụng và xác nhận các thẻ mới như `<article>` `<section>` , `<header>` , và `<footer>`...

B. HTML5 và các API mới

HTML5 bổ sung rất nhiều API mới mà lập trình viên có thể sử dụng để hỗ trợ cho các ứng dụng game của mình. Ví dụ như lưu lại dữ liệu với Web Storage, Web Sql, Indexed DB, thực hiện công việc song song với Web Worker, giao tiếp với server thông qua Web Socket. Do thời lượng có lượng, tôi chỉ trình bày một phần nhỏ trong số này.

I. Web Storage (DOM Storage)

HTML5 cung cấp một tính năng lưu trữ dữ liệu tại client với dung lượng giới hạn lớn hơn nhiều so với cookie. Tính năng này được gọi là Web Storage và được chia thành hai đối tượng là localStorage và sessionStorage. Bài viết này sẽ giúp bạn nắm được các kiến thức đầy đủ về sử dụng hai đối tượng này trong việc lập trình web.

1. Giới thiệu

Hiện nay, mỗi cookie chỉ cho phép lưu trữ tối đa 4KB và vài chục cookie cho một domain. Vì thế cookie chỉ được dùng để lưu trữ những thông tin đơn giản và ngắn gọn như email, username, … giúp người dùng đăng nhập tự động vào trang web. Điều này khiến cho những trang web muốn nâng cao hiệu suất làm việc bằng cách cache dữ liệu tại client hầu như không thể thực hiện được.

Sự xuất hiện của Web Storage là một điểm nhấn cho việc ra đời các ứng dụng web có khả năng tương tác và nạp dữ liệu tức thì trên trình duyệt. Một hiệu quả nữa là dung lương truyền tải qua mạng có thể được giảm thiểu đáng kể. Ví dụ một ứng dụng tra cứu sách trực tuyến, các sách đã được tra sẽ được lưu lại trên máy của người dùng. Khi cần tra lại, máy người dùng sẽ không cần kết nối đến server để tải lại những dữ liệu cũ.

Với những ứng dụng web có cơ sở dữ liệu nhỏ gọn, lập trình viên có thể thực hiện việc cache "âm thầm" cơ sở dữ liệu xuống client và sau đó người dùng có thể thoải mái tra cứu mà không cần request đến server.

2. Interface Storage

```
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();
};
```

Như bạn thấy, các dữ liệu lưu trữ trong Storage chỉ là kiểu chuỗi, với các loại dữ liệu khác số nguyên, số thực, bool, … bạn cần phải thực hiện chuyển đổi kiểu. Mỗi đối tượng Storage là một danh sách các cặp key/value, đối tượng này bao gồm các thuộc tính và phương thức:

- length: số lượng cặp key/value có trong đối tượng.
- key(n): trả về tên của key thứ n trong danh sách.
- getItem(key): trả về value được gắn với key.
- setItem(key,value): thêm hoặc gán một cặp key/value vào danh sách.
- removeItem(key): xóa cặp key/value khỏi danh sách.
- clear: xóa toàn bộ dữ liệu trong danh sách.

Bên cạnh đó, đối tượng Storage còn có thể được truy xuất qua các thuộc tính là các key trong danh sách.

Ví dụ:

```
localStorage.abc="123";
// equivalent to:
// localStorage.setItem("abc","123");
```

3. Local Storage và Session Storage

Hai đối tượng này là được tạo ra từ interface Storage, bạn sử dụng hai đối tượng này trong javascript qua hai biến được tạo sẵn là window.localStorage và window.sessionStorage. Hai lợi ích mà chúng mang lại là:

- Dễ sử dụng: bạn có thể truy xuất dữ liệu của hai đối tượng này thông qua các thuộc tính hoặc các phương thức. Dữ liệu được lưu trữ theo từng cặp key/value và không cần bất kì công việc khởi tạo hay chuẩn bị nào.
- Dung lượng lưu trữ lớn: Tùy theo trình duyệt, bạn có thể lưu trữ từ 5MB đến 10MB cho mỗi domain. Theo Wikipedia thì con số này là: 5MB trong Mozilla Firefox, Google Chrome,và Opera, 10MB trong Internet Explorer.

Phạm vi:

- 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 lưu trong sessionStorage hiện tại cũng bị xóa.

- localStorage: có thể truy xuất lẫn nhau giữa các cửa sổ trình duyệt. Dữ liệu sẽ được lưu trữ không giới hạn thời gian.

Đối với localStorage:

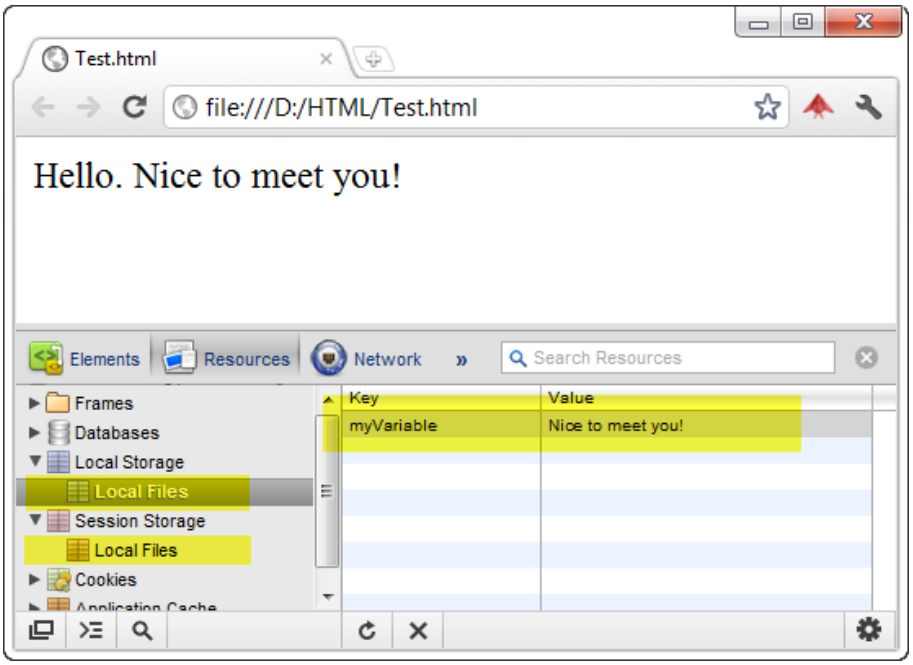
"Each domain and subdomain has its own separate local storage area. Domains can access the storage areas of subdomains, and subdomains can access the storage areas of parent domains. For example, `localStorage['example.com']` *is accessible to* `example.com` *and any of its subdomains. The subdomain* `localStorage['www.example.com']` *is accessible to* `example.com`*, but not to other subdomains, such as* `mail.example.com`*."*
http://msdn.microsoft.com/en-us/library/cc197062(VS.85).aspx

4. Sử dụng

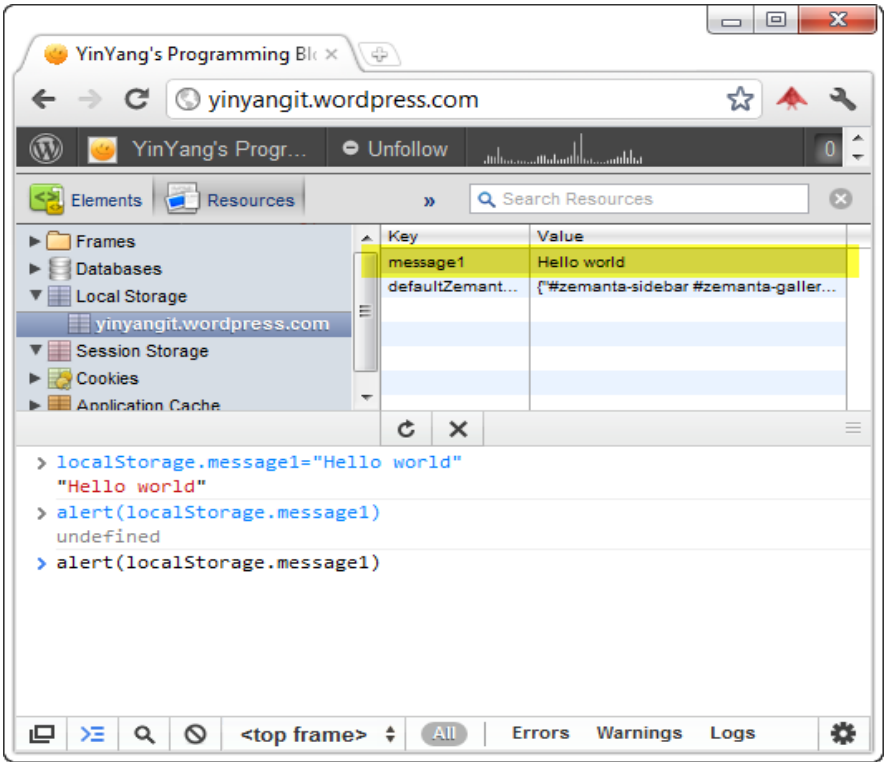
Bạn có thể tạo một tập tin HTML với nội dung phía dưới để chạy trên trình duyệt. Ở đây ta sử dụng Chrome vì nó cung cấp sẵn cửa sổ xem phần Resources trong Google Chrome Developer Tools (Ctrl + Shift + I). Nội dung của tập tin HTML như sau:

```
<!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>
```

Kết quả hiển thị:



Trong giao diện xem Resources, bạn có thể mở phần Console để gõ các lệnh javascript tương tác với trang web hiện tại. Ví dụ ở đây ta thêm các giá trị mới vào trong localStorage và dùng alert() để hiển thị chúng lên.



5. Storage event

Đối tượng Window trong javascript cung cấp một event với tên "storage". Event này được kích hoạt mỗi khi storageArea bị thay đổi.

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

Event này có thể không hoạt động khi bạn xem tập tin HTML ở máy cục bộ và chỉ được kích hoạt ở những cửa sổ/thẻ khác. Tức là khi bạn mở nhiều cửa sổ trình duyệt truy xuất đến cùng 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.

```
<!DOCTYPE html>
<html>
<body>

<button onclick="changeValue();">Change Value</button>

<script type="text/javascript">
localStorage.clear();
console.log(localStorage);
if (window.addEventListener)
        window.addEventListener('storage', storage_event, false);
else if (window.attachEvent) // IE
        window.attachEvent('onstorage', storage_event, false);

function storage_event(event){
        console.log(event);
}

function changeValue()
{
        localStorage.myValue=Math.random();
}

</script>

</body>
</html>
```

6. Thêm các phương thức vào Storage

Các phương thức của Storage có thể không đủ đáp ứng yêu cầu của bạn, vì vậy bạn có thể thêm một vài phương thức mới vào để sử dụng khi cần thiết. 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;
}
```

Phương thức trên sử dụng biểu thức chính quy để tìm kiếm theo hai tùy chọn “g” (tìm kiếm toàn bộ chuỗi) và “i” (không phân biệt hoa thường). Phương thức string.search() trả về vị trí của kí tự đầu tiên khớp với từ khóa tìm kiếm, ngược lại là -1 nếu không tìm thấy.

II. Web SQL Database

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.

1. Giới thiệu

Lợi ích của SQLite là có để được tích hợp vào các ứng dụng với một thư viện duy nhất để truy xuất được database. Chính vì vậy, bạn có thể sử dụng nó làm cơ sở dữ liệu cho những ứng dụng nhỏ và không cần thiết cài đặt bất kì phần mềm hoặc driver nào. Hiện tại Web SQL Database được hỗ trợ trong các trình duyệt Google Chrome, Opera và Safari.

Trong javascript, bạn sử dụng các phương thức chính sau để làm việc với Web SQL Database.

openDatabase: mở một database có sẵn hoặc tạo mới nếu nó chưa tồn tại.
transaction / readTransaction: hỗ trợ thực hiện các thao tác với database và rollback nếu xảy ra sai sót.
executeSql: thực thi một câu lệnh SQL.

2. Open Database

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

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

Tham số:
- name: tên của database.
- version: phiên bản. Hai database trùng tên nhưng khác phiên bản thì khác nhau.
- displayname: mô tả.
- estimatedSize: dung lượng được tính theo đơn vị byte.
- creationCallback: phương thức callback được thực thi sau khi database mở/tạo.

Ví dụ tạo một database với tên "mydb" và dung lượng là 5 MB:

```
var db = openDatabase("mydb", "1.0", "My First DB", 5 * 1024 * 1024);
```

3. Transaction

Bạn cần thực thi các câu SQL trong ngữ cảnh của một transaction. Một transaction cung cấp khả năng rollback khi một trong những câu lệnh bên trong nó thực thi thất 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.

Interface Database hỗ trợ hai phương thức giúp thực hiện điều này là transaction() và readTransaction(). Điểm khác biệt giữa chúng là transaction() hỗ trợ read/write, còn readTransaction() là read-only. Như vậy sử dụng readTransaction() sẽ cho hiệu suất cao hơn nếu như bạn chỉ cần đọc dữ liệu.

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

Ví dụ:

```
var db = openDatabase("mydb", "1.0", "My First DB", 5 * 1024 * 1024);
db.transaction(function (tx) {
  // Using tx object to execute SQL Statements
});
```

4. Execute SQL

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

Ví dụ 1: tạo bảng Customers và thêm hai dòng dữ liệu vào bảng này.

```
db.transaction(function (tx) {
  tx.executeSql("CREATE TABLE IF NOT EXISTS CUSTOMERS(id unique, name)");
  tx.executeSql("INSERT INTO CUSTOMERS (id, name) VALUES (1, 'peter')");
  tx.executeSql("INSERT INTO CUSTOMERS (id, name) VALUES (2, 'paul')");
});
```

Tuy nhiên cách thực thi SQL trên không được rõ ràng và có thể bị các vấn đề về bảo mật như SQL injection. Vì vậy tốt hơn bạn nên để các tham số cần truyền cho câu SQL trong một mảng và đặt vào làm tham số thứ 2 của phương thức executeSql(). Các vị trí trong câu SQL chứa tham số sẽ được thay thế bởi dấu '?':

```
tx.executeSql("INSERT INTO CUSTOMERS (id, name) VALUES (?,?)", [1,"peter"]);
tx.executeSql("INSERT INTO CUSTOMERS (id, name) VALUES (?,?)", [2,"paul"]);
```

III. Web Worker

Với công nghệ phần cứng hiện nay, việc sử dụng đa luồng đã trở nên một phần không thể thiếu trong các phần mềm. Tuy nhiên, công nghệ thiết kế web vẫn chưa tận dụng được sức mạnh này. Với các công việc đòi hỏi một quá trình xử lý lâu, lập trình viên thường phải sử dụng những thủ thuật như setTimeout(), setInterval(),… để thực hiện từng phần công việc. Hiện nay, để giải quyết vấn đề này, một API mới dành cho javascript đã xuất hiện với tên gọi Web Worker.

1. Giới thiệu

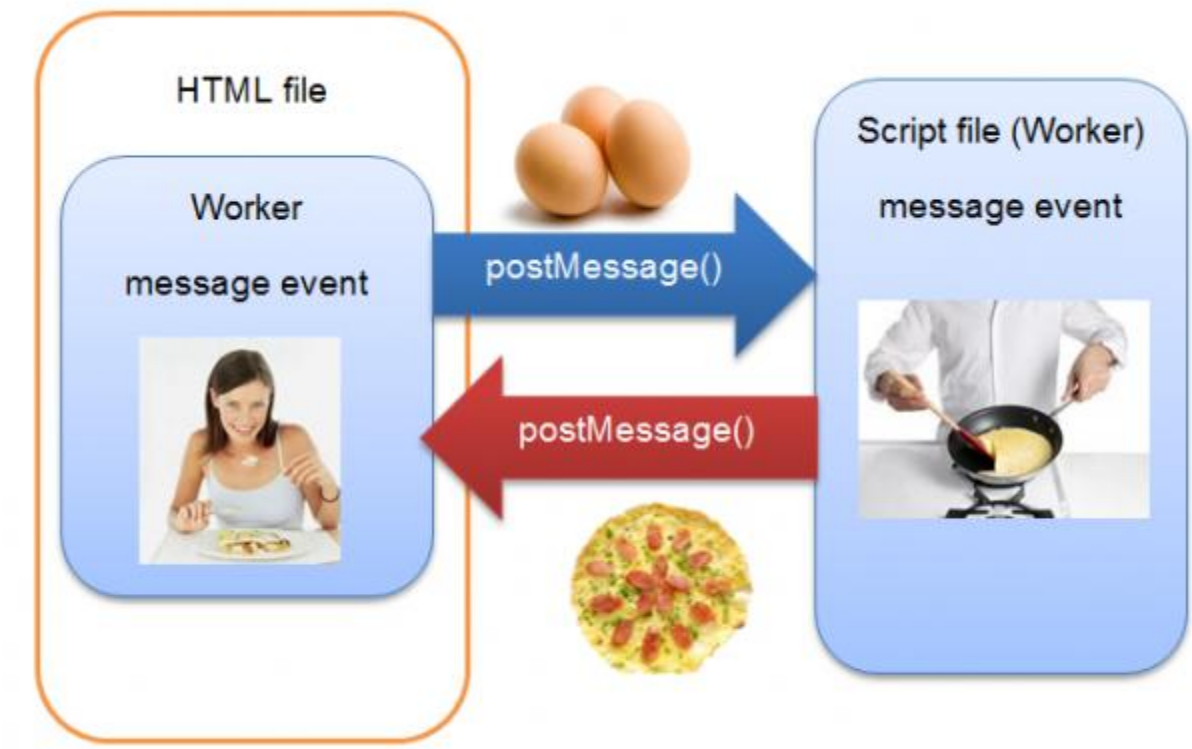
Đối tượng Web Worker được tạo ra sẽ thực thi trong một thread độc lập và chạy ở chế độ nền nên không ảnh hưởng đến giao diện tương tác của trang web với người dùng. Với đặc điểm này, bạn có thể sử dụng Web Workert các công việc đòi hỏi thời gian xử lý lâu nạp dữ liệu, tạo cache,…

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.

Việc tạo một Web Worker sẽ cần thực hiện như sau:

```
var worker = new Worker('mytask.js');
```

Tập tin script sẽ được tải về và Worker chỉ được thực thi sau khi tập tin này tải hoàn tất. Trong tập tin script này, ta sẽ xử lý sự kiện ‘message’ của Worker từ các dữ liệu nhận được thông qua phương thức postMessage(). Phương thức này chấp nhận một tham số chứa thông điệp cần gửi đến tập tin script để xử lý. Dữ liệu này sẽ được lấy thông qua thuộc tính data của tham số event trong hàm xử lý sự kiện message. Quy trình này được mô tả trong hình sau:



2. Ví dụ đơn giản nhất:

Tạo hai tập tin sau trong cùng một thư mục:

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 />
<button id="button1">Submit</button>
<script>

    worker = new Worker("mytask.js");

    worker.onmessage = function (event) {
       alert(event.data);
    };

    document.getElementById("button1").onclick = function() {
```

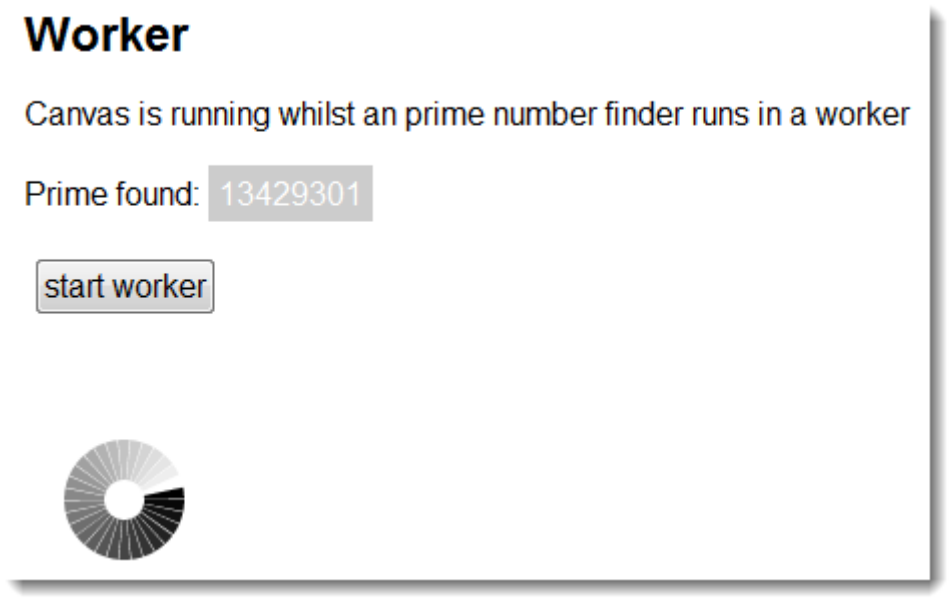
```
        var name = document.getElementById("username").value;

        worker.postMessage(name);
    };
</script>
</body>
</html>
```

Bây giờ bạn chạy thử và nhấn nút Submit, một thông điệp sẽ hiển thị với nội dung tương tự "Hello Yin Yang".

3. Kết luận

Với các công việc đơn giản, lập trình viên sẽ gửi đi một dữ liệu kiểu mảng bao gồm tên lệnh và các dữ liệu cần xử lý. Worker sẽ phân tích dữ liệu nhận được và gọi các phương thức xử lý tương ứng. Tuy nhiên, một Worker bạn tạo ra chỉ nên dành riêng để thực hiện một công việc cụ thể. Bởi vì mục đích chính của việc tạo ra chúng là để làm những công việc "nặng nhọc". Cuối cùng, khi đã hoàn tất công việc, bạn hãy giải phóng cho Worker bằng cách dùng phương thức worker.terminate().

Bạn có thể xem demo sau về cách sử dụng Web Worker để thực hiện đồng thời việc tính toán tìm các số nguyên tố và cập nhật lại canvas:



IV. Tạo chuyển động với WindowAnimationTiming API

Thay vì đặt timeout để gọi các phương thức vẽ lại hình ảnh, cách tốt nhất mà bạn nên sử dụng để tạo các hiệu ứng chuyển động trong canvas là dùng API WindowAnimationTiming, thông qua phương thức chính là requestAnimationFrame().

1. setTimeout và setInterval

Cách truyền thống mà bạn dùng để tạo các hiệu ứng đồ họa chuyển động với javascript là gọi liên tục công việc update và draw sau những khoảng thời gian xác định thông qua phương thức setInterval() hoặc setTimeout(). Mỗi lần gọi, một frame (khung hình) mới sẽ được tạo ra và vẽ đè lên màn hình cũ.
Khó khăn của phương pháp này là khó xác định được giá trị thời gian thích hợp dựa trên mỗi thiết bị được sử dụng (thông thường khoảng 60 fps – frames per second). Ngoài ra với những hiệu ứng phức tạp thì việc update/draw có thể diễn ra lâu hơn so với thời gian giữa hai lần gọi.

Cách tổng quát để giải quyết vấn đề trên là thực hiện tính toán dựa vào khoảng cách thời gian giữa lần gọi trước đó và hiện tại, sau đó xác định bỏ qua một vài bước draw hoặc thay đổi giá trị timeout cho phù hợp.

2. WindowAnimationTiming

Một tính năng mới ra đời cho phép bạn đơn giản hóa công việc này là API WindowAnimationTiming.

Đây là một interface bao gồm hai phương thức là requestAnimationFrame() và cancelAnimationFrame(). Việc xác định thời điểm cập nhật 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);
```

requestAnimationFrame: gửi request đến trình duyệt thực hiện một hành động update/draw frame mới thông qua tham số callback. Các request này sẽ được lưu trong đối tượng document với các cặp <handle, callback> và được thực hiện lần lượt. Giá trị handle là một số định danh được tạo ra và trả về sau khi gọi phương thức này.
cancelAnimationFrame: hủy một request được tạo ra requestAnimationFrame với tham số là handle của request.
Ngoài ra còn có thuộc tính window.mozAnimationStartTime (chỉ mới được hỗ trợ trên Firefox) chứa giá trị milisecond là khoảng cách từ mốc thời gian (1/1/1970) đến thời điểm bắt đầu của request cuối cùng được thực hiện. Giá trị này tương đương với giá trị trả về của Date.now() hoặc Date.getTime(), mốc thời gian là bao nhiêu không quan trọng nhưng nó giúp bạn biết được khoảng thời gian giữa hai lần thực hiện request.

3. Lợi ích và hiệu quả

Microsoft cho ta một ví dụ trực quan về hiệu quả của requestAnimationFrame() so với setTimeout() tại trang requestAnimationFrame API. Qua ví dụ này, ta thấy được số lần gọi callback (update) thực tế của setTimeout() lớn hơn so với dự tính. Ngoài ra các kết quả cho thấy hiệu suất của việc thực thi callback, CPU, mức tiêu tốn năng lượng và ảnh hưởng đến các tác vụ nền của requestAnimationFrame() hơn hẳn so với setTimeout():

requestAnimationFrame based animations

Actual callbacks:	582
Expected callbacks:	582
Callback Efficiency:	100%
CPU Efficiency:	High
Power Consumption:	Low
Background Interference:	Low

	setTimeout	requestAnimationFrame
Expected callbacks	40543	40544
Actual callbacks	41584	40544
Callback Efficiency	59.70%	100.00%
Callback Efficiency	Medium	High
Power Consumption	Medium	Low
Background Interference	High	Low

Một hiệu quả khác các animation sẽ tự động dừng lại nếu tab chứa nó không được hiển thị (khi bạn chuyển sang một tab khác của trình duyệt). Điều này giúp hạn chế được tài nguyên sử dụng một cách hợp lý.

4. Sử dụng

Kiểm tra trình duyệt hỗ trợ:

Tùy theo trình duyệt mà tên của phương thức này sẽ có những prefix khác nhau (vendor):

```
_reqAnimation = window.requestAnimationFrame  ||
                        window.mozRequestAnimationFrame       ||
                        window.webkitRequestAnimationFrame    ||
                        window.msRequestAnimationFrame        ||
                        window.oRequestAnimationFrame         ;

if(_reqAnimation)
        update();
else
        alert("Your browser doesn't support requestAnimationFrame.");
```

Ví dụ hoàn chỉnh:

```
<HTML>
<head>
  <script>
        const RADIUS = 20;
        var cx =  100;
        var cy = 100;
        var speedX = 2;
        var speedY = 2;
        var _canvas;
        var _context;
        var _reqAnimation;
        var _angle = 0;

        function update(time) {

                cx += speedX;
                cy += speedY;
```

```
                if(cx<0 || cx > _canvas.width)
                        speedX = -speedX;

                if(cy<0 || cy > _canvas.height)
                        speedY = -speedY;

                // draw
                _context.clearRect(0, 0, _canvas.width, _canvas.height);
                _context.beginPath();
                _context.arc(cx, cy, RADIUS, 0, Math.PI*2, false);
                _context.closePath();
                _context.fill();

                _reqAnimation(update);
        }
        window.onload = function(){
                _canvas  = document.getElementById("canvas");
                _context = _canvas.getContext("2d");
                _context.fillStyle = "red";
                cx = _canvas.width/2;
                cy = _canvas.height/2;

        _reqAnimation = window.requestAnimationFrame  ||
                        window.mozRequestAnimationFrame               ||
                        window.webkitRequestAnimationFrame    ||
                        window.msRequestAnimationFrame        ||
                        window.oRequestAnimationFrame         ;

        if(_reqAnimation)
                update();
        else
                alert("Your browser doesn't support requestAnimationFrame.");
                };
  </script>
</head>
<body>
<canvas id="canvas" width="400px" height="300px" style="border: 1px
solid"></canvas>
</body>
</HTML>
```

C. Canvas 2D API

HTML5 xác định <canvas> như một bitmap phụ thuộc vào độ phân giải, được sử dung để vẽ đồ thị, đồ họa game hoặc hình ảnh trực quan khác. Canvas là hình chữ nhật trên trang và có thể sử dụng JavaScript để vữ bất cứ điều gì bạn muốn. HTML5 định nghĩa một tập các chức năng (canvas API) để vẽ hình dạng, xác định đường dẫn, tạo gradient.

HTML5 Canvas là một JavaScript API để mã hóa các bản vẽ. Canvas API cho phép định nghĩa một đối tượng canvas như là thử <canvas> trên trang HTML ,chúng ta có thể vẽ bên trong nó. <canvas> là một khối không gian vô hình, mặc định là 300x250 pixels (trên tất cả trình duyệt)

Chúng ta có thể vẽ cả hình 2D và 3D (WebGL). 2D có sẵn trong tất cả các trình duyệt Web hiện đại, IE9, và thông qua thư viện excanvas.js trong các phiên bản IE hiện tại.

Canvas 2D cung cấp một API đơn giản nhưng mạnh mẽ để có thể vẽ một cách nhanh chóng trên bề mặt bitmap 2D. Không có định dạng tập tin, và bạn chỉ có thể vẽ bằng cách sử dụng script. Bạn không có bất kỳ các nút DOM cho các hình khối bạn vẽ - bạn đang vẽ pixels, không phải vectơ và chỉ là các điểm ảnh được ghi nhớ.

Một số tính năng cả Canvas:

- Vẽ hình ảnh
- Tô màu
- Tạo hình học và các kiểu mẫu
- Văn bản
- Sao chép hình ảnh, video, những canvas khác
- Thao tác điểm ảnh
- Xuất nội dung của một thẻ <canvas> sang tập tin ảnh tĩnh.

I. Vẽ ảnh và thao tác với pixel

Muốn tạo ra những hiệu ứng đồ họa đặc biệt khi sử dụng canvas, bạn không thể chỉ sử dụng cá thuộc tính và phương thức có sẵn của đối tượng context. Chính vì vậy, bài viết này sẽ giới thiệu cho bạn cách vẽ ảnh và thao tác với các pixel từ đối tượng ImageData.

1. Nạp và vẽ ảnh

Để vẽ một ảnh ra canvas, ta tạo một đối tượng image và thực hiện phương thức context.drawImage() trong sự kiện load của image. Như vậy để đảm bảo rằng hình ảnh chỉ được vẽ sau khi đã được nạp hoàn tất. Ngoài ra, bạn nên đặt sự kiện load trước khi gán đường dẫn cho ảnh. Nếu không image có thể được load xong trước khi bạn gắn sự kiện load cho nó.

Phương thức drawImage() có ba overload sau:

- Vẽ image tại một vị trí destX, destY:

```
context.drawImage(image,destX,destY);
```

- Vẽ image tại vị trí destX, destY và kích thước destWidth, destHeight:

```
context.drawImage(image,destX,destY,destWidth,destHeight);
```

- Cắt image tại vị trí [sourceX, sourceY, sourceWidth, sourceHeight] và vẽ tại [destX, destY, destWidth, destHeight]:

```
context.drawImage(image, sourceX, sourceY, sourceWidth,

sourceHeight, destX, destY, destWidth, destHeight);
```

Ví dụ:

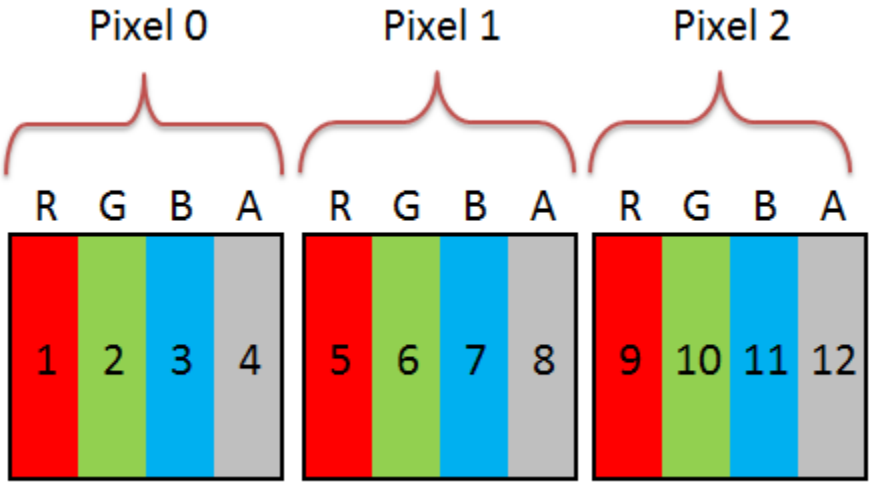
```
window.onload = function(){

        var canvas = document.getElementById("mycanvas");
        var context = canvas.getContext("2d");

        var img = new Image();

        img.onload = function(){
                context.drawImage(img, 10, 10,50,50);
        };
        img.src = "foo.png";
};
```

2. Thao tác với pixel

Một ảnh bao gồm một mảng các pixel với các giá trị red, green, blue và alpha (RGBA). Trong đó alpha là giá trị xác định độ mờ đục (opacity) của ảnh. Giá trị alpha càng lớn thì độ màu sắc càng rõ nét và màu sắc sẽ trở nên trong suốt nếu alpha là 0.

Trong Canvas 2D API, dữ liệu ảnh được lưu trong một đối tượng ImageData với 3 thuộc tính là width, height và data. Trong đó data là một mảng một chiều chứa các pixel. Mỗi pixel chứa 4 phần tử tương ứng là R,G,B,A.

Như vậy với một ảnh có kích thước 10×20 ta sẽ có 200 pixel và có 200*4=400 phần tử trong mảng ImageData.data.



Bạn có thể tham khảo thông tin về các API này tại: http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.HTML#pixel-manipulation:

imagedata = context.createImageData(sw, sh)

Trả về một đối tượng ImageData với kích thước sw x sh. Tất cả pixel của đối tượng này có màu đen trong suốt.

imagedata = context.createImageData(imagedata)
Trả về đối tượng ImageData với kích thước bằng với đối tượng trong tham số. Tất cả pixel có màu đen trong suốt.
imagedata = context.getImageData(sx, sy, sw, sh)

Trả về một đối tượng ImageData chứa dữ liệu ảnh vùng chữ nhật (xác định bởi các tham số) của canvas.

Ném NotSupportedError exception nếu như có bất kì tham số nào không phải là số hợp lệ. Ném IndexSizeError exception nếu width hoặc height là zero.

imagedata.width
imagedata.height

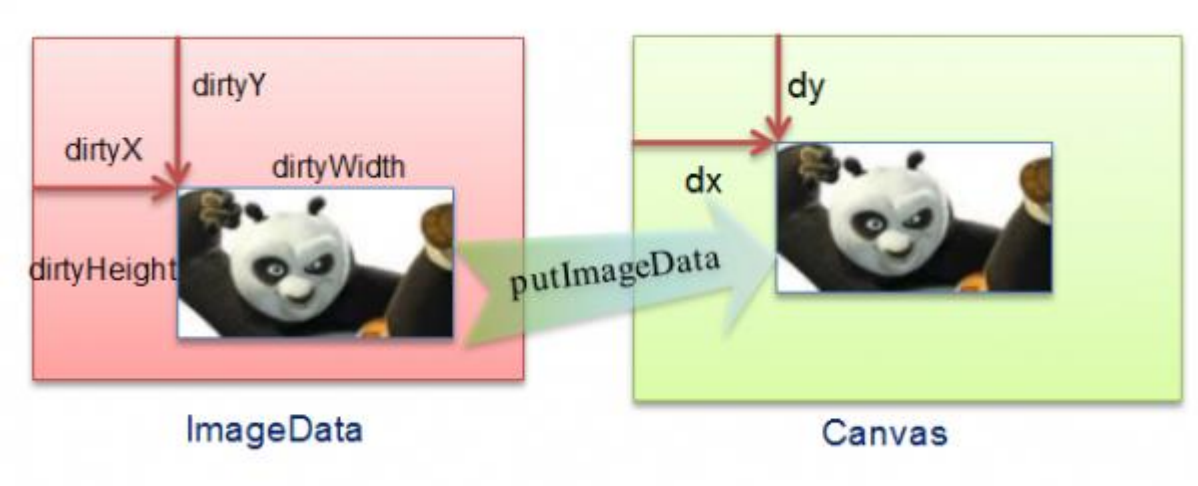
Trả về kích thước thật của đối tượng ImageData, tính theo pixel.

imagedata.data

Trả về mảng một chiều chứa dữ liệu dạng RGBA, mỗi giá trị nằm trong khoảng 0 đến 255.

context . putImageData(imagedata, dx, dy [, dirtyX, dirtyY, dirtyWidth, dirtyHeight])

Vẽ dữ liệu từ đối tượng ImageData lên canvas tại vị trí dx, dy. Nếu như hình chữ nhật (từ các tham số dirtyX, dirtyY, dirtWidth, dirtyHeight) được xác định, thì phần dữ liệu của ImageData trong vùng chữ nhật này mới được vẽ lên canvas.



Các thuộc tính xác định hiệu ứng vẽ của context sẽ bị bỏ qua khi phương thức này được gọi. Các pixel từ canvas sẽ được thay thế hoàn toàn bởi ImageData mà không có các sự kết hợp màu sắc, hiệu ứng, … với các dữ liệu ảnh sẵn có trên canvas.

Một trong những ví dụ thường gặp và đơn giản nhất là đảo ngược màu của ảnh. Điều này được thực hiện bằng cách lấy giá trị màu tối đa (255) trừ đi giá trị của mỗi kênh màu RGB hiện tại của mỗi pixel. Giá trị alpha sẽ để giá trị tối đa để ảnh được rõ nét nhất.

```
<HTML>
<head>
<script>
        window.onload = function(){

                var img = new Image();
                img.onload = function(){
                        invertColor(this);
                };
                img.src="panda.jpg";
        };
        function invertColor(img){
                var canvas = document.getElementById("mycanvas");
                var context = canvas.getContext("2d");

                // draw image at top-left corner
                context.drawImage(img,0,0);
                // draw original image right beside the previous image
                context.drawImage(img,img.width,0);

                // get ImageData object from the left image
                var imageData = context.getImageData(0, 0, img.width,
img.height);
                var data = imageData.data;
```

```
                for (var i = 0; i < data.length; i += 4) {

                        data[i]         =       255 - data[i]; // red
                        data[i + 1]     =       255 - data[i+1]; // green
                        data[i + 2]     =       255 - data[i+2]; // blue
                        data[i + 3]     =       255; // alpha
                }

                context.putImageData(imageData,0,0);

        }
</script>
</head>
<body>
   <canvas id="mycanvas" width="600" height="250"></canvas>
</body>
</HTML>
```

Kết quả:

Bạn có thể thêm các tham số để tạo một "dirty rectangle" trong phương thức putImageData() nếu muốn vẽ ImageData lên một vùng chữ nhật xác định của canvas.

Ví dụ ta chọn cùng một vùng chữ nhật trên ImageData và vẽ lên hai vùng chữ nhật khác nhau trên canvas để được kết quả sau:

```
context.putImageData(imageData,0,0,0,0,img.width/2,img.height/2);

context.putImageData(imageData,img.width,0,0,0,img.width/2,img.height/2);
```

II. Vẽ hình bằng chuột

Trong phần này, ta sẽ tìm hiểu cách bắt và xử lý các thao tác chuột trên Canvas để thực hiện một ứng dụng vẽ hình đơn giản. Bên cạnh đó, bạn có thể biết được cách để lưu và phục hồi lại dữ liệu của canvas khi cần thiết.

1. Xác định tọa độ chuột

Để xác định được tọa chuột trên canvas, ta sẽ sử dụng tham số của các sự kiện như mousedown, mousemove, … Tham số của các sự kiện này chứa tọa độ chuột lưu trong hai biến pageX và pageY. Ta sẽ dùng tọa độ này trừ đi tọa độ của canvas (canvas.offsetLeft và canvas.offsetTop) để lấy được vị trí chính xác của chuột trên canvas.

Đầu tiên là việc mô phỏng công cụ Pen cho phép vẽ tự do trong canvas:

```
<HTML>
<head>
<script src="http://code.jquery.com/jquery-latest.js"></script>
<script>
var preX;
var preY;
var paint;
var canvas;
var context;

$(function(){
        canvas = document.getElementById("canvas");
        context = canvas.getContext("2d");

        $('#canvas').mousedown(function(e){
                preX = e.pageX - this.offsetLeft;
                preY = e.pageY - this.offsetTop;

                paint = true;
        });
        $('#canvas').mousemove(function(e){
```

```
          if(paint){
                var x = e.pageX - this.offsetLeft;
                var y = e.pageY - this.offsetTop;
                context.moveTo(preX,preY);
                context.lineTo(x,y);
                context.stroke();
                preX = x;
                preY = y;
          }
        });
        $('#canvas').mouseenter(function(e){
                if(paint)
                {
                preX = e.pageX - this.offsetLeft;
                preY = e.pageY - this.offsetTop;
                }
        });
        $("#canvas").mouseup(function(){
                paint = false;
        });
        $('#canvas').mouseleave(function(e){
                paint = false;
        });

});

</script>
</head>
<body>
   <canvas id="canvas" width="500px" height="500px" style="border: 1px solid
gray;"></canvas>
</body>
</HTML>
```

2. Lưu nội dung của Canvas

Để lưu lại dữ liệu của Canvas nhằm phục hồi khi cần thiết (chẳng hạn như chức năng Undo, Redo), ta sẽ sử dụng phương thức context.getImageData().Ta sẽ sử dụng phương pháp này để thực hiện chức năng vẽ đoạn thẳng trên Canvas. Trước khi bắt đầu vẽ một đoạn thằng, canvas cần được lưu lại nội dung và mỗi khi chuột di chuyển, ta sẽ phục hồi lại nội dung được lưu đó đồng thời vẽ một đoạn thẳng từ điểm bắt đầu đến vị trí chuột.

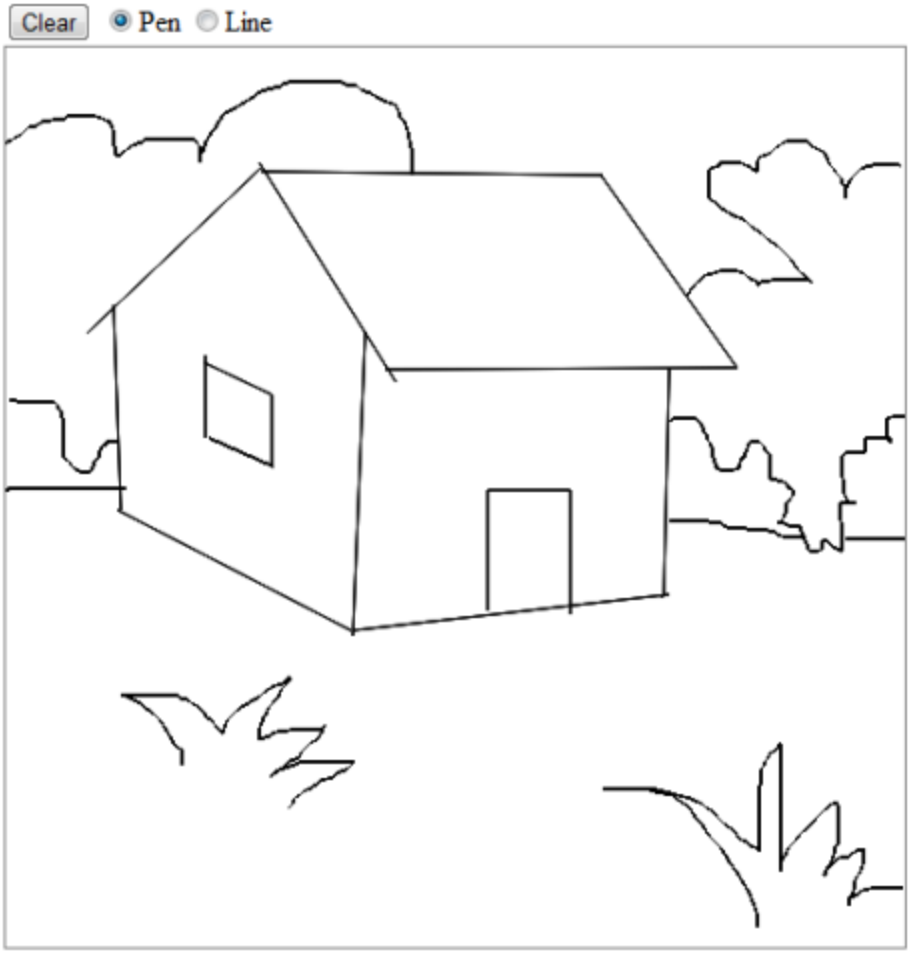
Ta cũng sửa ví dụ trên thành một jQuery plugin để tiện sử dụng:

```
<HTML>
<head>

<script src="http://code.jquery.com/jquery-latest.js"></script>
<script>

(function( $ ) {

        var preX;
        var preY;
```

```
        var tool;       // pen, line
        var canvas;
        var context;
        var imageData;
        var paint;

  $.fn.makeDrawable = function() {

        canvas = this[0];
        if( !$(canvas).is("canvas") )
                throw "The target element must be a canvas";

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

        $(canvas).mousedown(function(e){
                preX = e.pageX - canvas.offsetLeft;
                preY = e.pageY - canvas.offsetTop;
                paint = true;
                if(tool=="line")
                {
                        imageData  =  context.getImageData(0,  0,  canvas.width,
canvas.height);
                }
        });
        $(canvas).mousemove(function(e){
                if(paint)
                {
                        var x = e.pageX - canvas.offsetLeft;
                        var y = e.pageY - canvas.offsetTop;

                        if(tool=="pen")
                        {
                                context.moveTo(preX,preY);
                                context.lineTo(x,y);
                                context.stroke();

                                preX = x;
                                preY = y;
                        }
                        else if(tool=="line")
                        {
                                canvas.width=canvas.width;   //   clear   canvas
content
                                context.putImageData(imageData,0,0);

                                context.moveTo(preX,preY);
                                context.lineTo(x,y);
                                context.stroke();
                        }

                }
        });

        $(canvas).mouseup(function(e){
                if(tool=="line")
                {
                        var x = e.pageX - canvas.offsetLeft;
```

```
                        var y = e.pageY - canvas.offsetTop;

                        context.moveTo(preX,preY);
                        context.lineTo(x,y);
                        context.stroke();
                }
                paint = false;
        });
        $(canvas).mouseleave(function(e){
                paint = false;
        });

        return $(canvas);
  };

  $.fn.setTool = function(newTool) {
        tool=newTool;
        return $(canvas);
  }
 $.fn.clear = function() {
        canvas.width=canvas.width;
        return $(canvas);
  }
})( jQuery );

$(function(){

        $("#canvas").makeDrawable();
        $("#button1").click(function(){
                $("#canvas").clear();
        });

        $("#pen").change(function(){
                if(this.value)
                        $("#canvas").setTool("pen");
        });
        $("#line").change(function(){
                if(this.value)
                        $("#canvas").setTool("line");
        });

        $("#canvas").setTool("pen");
});

</script>
</head>
<body>
        <div>
        <button id="button1">Clear</button>
        <input name="tool" type="radio" id="pen" checked="true">Pen</input>
        <input name="tool" type="radio" id="line">Line</input>
        </div>
   <canvas id="canvas" width="500px" height="500px" style="border: 1px solid
gray;"></canvas>
</body>
</HTML>
```

Kết quả:



III. Chọn và di chuyển đối tượng

Thay vì lưu trữ nội dung của Canvas dưới dạng ImageData, ta có thể lưu trữ các đối tượng đồ họa dưới dạng cấu trúc dữ liệu và thực hiện vẽ từng phần tử lên Canvas. Với phương pháp này, ta sẽ minh họa bằng một ví dụ sử dụng chuột để chọn và di chuyển các hình vẽ trên Canvas.

1. Tạo cấu trúc dữ liệu

Đầu tiên ta sẽ tạo một lớp Rect dùng để chứa dữ liệu của một hình chữ nhật. Lớp này ngoài các biến lưu trữ tọa độ, kích thước và trạng thái (selected) thì cần một phương thức dùng để kiểm tra xem một tọa độ [x,y] có nằm bên trong nó không. Ta gọi phương thức này là isContain() và có kiểu trả về là boolean. Mã nguồn của lớp này có dạng:

```
function Rect() {
  this.isSelected = false;
  this.x = 0;
  this.y = 0;
  this.width = 1;
  this.height = 1;
}
Rect.prototype.isContain = function(x,y){
        var right = this.x + this.width;
        var bottom = this.y + this.height;
```

```
        return x > this.x && x < right &&
                y > this.y && y < bottom;
}
```

Tiếp theo, ta tạo một lớp ShapeList dùng để chứa các đối tượng Rect trong một kiểu dữ liệu mảng. Ngoài ra, ShapeList sẽ có thêm một biến dùng để chỉ ra đối tượng Rect đang được chọn (selectedItem), và hai biến dùng để lưu vị trí của chuột khi click lên một đối tượng Rect (offsetX, offsetY). Hai biến offset này dùng để tính tọa độ chính xác khi người dùng sử dụng chuột để di chuyển đối tượng Rect.

ShapeList còn có hai phương thức:

- **addItem:** thêm một đối tượng Rect vào danh sách.
- **selectAt:** tìm và chọn đối tượng Rect chứa tọa độ [x,y]. Ta sẽ duyệt ngược từ cuối mảng để đảm bảo các đối tượng nằm sau sẽ được chọn trước trong trường hợp nhiều Rect cùng chứa điểm [x,y]

```
function ShapeList(){
        this.items = [];
        this.selectedItem = null;
        this.offsetX = -1;
        this.offsetY = -1;
}

ShapeList.prototype.addItem = function(x,y,width,height){
        var rect = new Rect;
        rect.x = x;
        rect.y = y;
        rect.width = width;
        rect.height = height;

        this.items.push(rect);
}

ShapeList.prototype.selectAt = function(x,y){
        if(this.selectedItem)
                this.selectedItem.isSelected = false;
        this.selectedItem = null;
        for (var i = 0; i < this.items.length; i++) {
                var rect = this.items[i];
                if(rect.contains(x,y))
                {
                        this.selectedItem = this.items[i];
                        this.offsetX = x - this.items[i].x;
                        this.offsetY = y - this.items[i].y;
                        this.items[i].isSelected = true;
                        break;
                }
    }
}
```

2. Các phương thức vẽ bằng context

```
function draw(){
        clear();
```

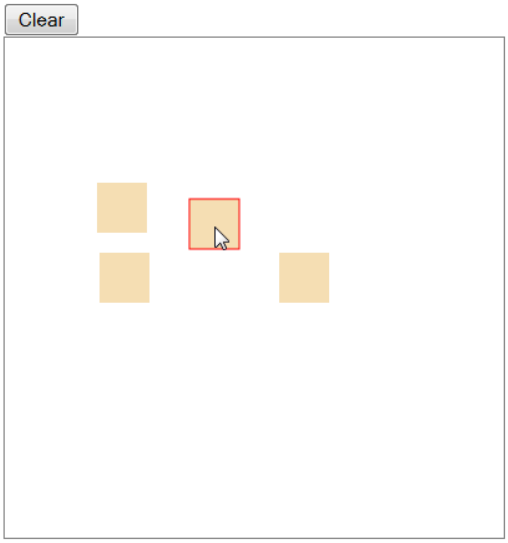
```
    for (var i = _list.items.length-1;i>=0; i--) {
                drawRect(_list.items[i]);
    }
}

function drawRect(rect){
        _context.fillRect(rect.x,rect.y,rect.width,rect.height);
        if(rect.isSelected)
        {
                _context.save();
                _context.strokeStyle = "red";
                _context.strokeRect(rect.x,rect.y,rect.width,rect.height);
                _context.restore();
        }
}
```

3. Các sự kiện chuột của Canvas

Trong các sự kiện này, ta sẽ dùng cờ _ismoving để xác định xem một đối tượng có đang được chọn hay không và thực hiện di chuyển đối tượng ShapeList.selectedItem trong mousemove. Giá trị _ismoving này sẽ được xác định trong sự kiện mousedown và bị hủy khi mouseup.

```
function canvas_mousedown(e){

        var x = e.pageX - _canvas.offsetLeft;
        var y = e.pageY - _canvas.offsetTop;

        _list.selectAt(x,y)

        if(!_list.selectedItem)
                _list.addItem(x-RECT_SIZE,y-
RECT_SIZE,RECT_SIZE*2,RECT_SIZE*2);

        _ismoving = true;
        draw();
}
function canvas_mousemove(e){
        if(_ismoving && _list.selectedItem){
                var x = e.pageX - _canvas.offsetLeft;
                var y = e.pageY - _canvas.offsetTop;

                _list.selectedItem.x = x - _list.offsetX;
                _list.selectedItem.y = y - _list.offsetY;

                draw();
        }
}
function canvas_mouseup(e){
        _ismoving = false;
}
```

Kết quả:



IV. Sử dụng bàn phím

Bàn phím là thiết bị không thể thiếu và là phương tiện rất quan trọng để thực hiện các chức năng của các ứng dụng tương tác với người dùng. Trong bài viết này, ta sẽ hướng dẫn cách bắt sự kiện bàn phím trong canvas và dùng nó để điều khiển góc xoay và hướng di chuyển của đối tượng đồ họa.

1. Bắt sự kiện bàn phím

Để bắt sự kiện này, bạn có thể đăng kí trực tiếp cho đối tượng window, tuy nhiên điều này sẽ gây ra những rắc rối khi trang của bạn có quá nhiều thành phần. Cách tốt nhất là đăng kí riêng cho canvas các hàm xử lý. Tuy nhiên, để canvas có thể focus được, bạn cần xác định thuộc tínhTabIndex của nó:

```
<canvas id=”canvas” width=”300″ height=”200″

tabindex=”1″ style=”border: 1px solid;”></canvas>
```

Sau đó bạn đăng kí các sự kiện cần thiết cho canvas. Khi chạy trên trình duyệt, bạn cần phải click chuột vào canvas để nó nhận được focus trước khi thực hiện các thao tác bàn phím.

```
_canvas.onkeydown = canvas_keyDown;

function canvas_keyDown(e){
        alert(e.keyCode);
}
```

Tham số event truyền vào hàm xử lý sẽ chứa các thuộc tính cần thiết để bạn xác định được phím nào được nhấn. Ở đây, bạn chỉ cần quan tâm đến thuộc tính keyCode lưu mã của phím được nhấn. Do các giá trị keyCode có kiểu số nên rất khó nhớ và kiểm tra, may mắn là ta tìm được

trang "Javascript Keycode Enums" liệt kê sẵn các keyCode dưới dạng enum của một đối tượng. Ta sửa lại một chút để tiện sử dụng hơn:

```
var Keys = {
      BACKSPACE: 8,
      TAB: 9,
      ENTER: 13,
      SHIFT: 16,
      CTRL: 17,
      ALT: 18,
      PAUSE: 19,
      CAPS_LOCK: 20,
      ESCAPE: 27,
      SPACE: 32,
      PAGE_UP: 33,
      PAGE_DOWN: 34,
      END: 35,
      HOME: 36,
      LEFT_ARROW: 37,
      UP_ARROW: 38,
      RIGHT_ARROW: 39,
      DOWN_ARROW: 40,
      INSERT: 45,
      DELETE: 46,
      KEY_0: 48,
      KEY_1: 49,
      KEY_2: 50,
      KEY_3: 51,
      KEY_4: 52,
      // ...
    };
```

2. Kiểm tra trạng thái của nhiều phím

Một khó khăn của các sự kiện bàn phím trong javascript là chỉ có thể xác định được duy nhất một phím nhấn thông qua thuộc tính event.keyCode. Để có thể kiểm tra được trạng thái của nhiều phím được nhấn cùng lúc, ta phải lưu trạng thái của chúng lại khi sự kiện keyDown xảy ra và xóa trạng thái đó đi trong sự kiện keyUp.

```
var _keypressed = {};
function canvas_keyDown(e){
        _keypressed[e.keyCode] = true;
}
function canvas_keyUp(e){
        _keypressed[e.keyCode] = false;
}
```

3. Giới hạn các phím được bắt

Sử dụng phương pháp trên, bạn cần lọc các phím cần sử dụng để đối tượng _keypressed không lưu các giá trị không cần thiết. Một cách đơn giản nhất là sử dụng mảng để lưu keyCode của các phím này và kiểm tra trong các sự kiện bàn phím:

```
const AVAILABLE_KEYS =
        [       Keys.UP_ARROW,
                Keys.DOWN_ARROW,
                Keys.LEFT_ARROW,
                Keys.RIGHT_ARROW
```

```
        ];
function canvas_keyDown(e){
        if(AVAILABLE_KEYS.indexOf(e.keyCode)!=-1)
        {
                _keypressed[e.keyCode] = true;
        }
}
function canvas_keyUp(e){
        if(_keypressed[e.keyCode])
        {
                _keypressed[e.keyCode] = false;
        }
}
```

V. Chuyển động trong Canvas

Một ví dụ đơn giản để khi làm quen với đồ họa và chuyển động trong lập trình là viết một ví dụ bóng nảy bên trong một vùng cửa sổ (canvas). Một quả bóng sẽ được vẽ bên trong canvas và chuyển động theo một hướng xác định. Khi chạm bất kì thành tường nào, bóng sẽ đổi hướng chuyển động tùy theo hướng di chuyển.

1. Cơ bản

Ta xác định hai giá trị speedX và speedY tương ứng với tốc độ di chuyển theo phương ngang và dọc của trái bóng. Hướng di chuyển của bóng phụ thuộc vào giá trị âm hoặc dương của speedX và speedY.

Trong ví dụ này, thành tường là các biên của canvas và chỉ có hai phương là ngang và dọc. Nguyên lý hoạt động rất đơn giản: khi bóng tiếp xúc với biên dọc của canvas, ta sẽ đổi chiều của speedY và với biên ngang ta sẽ đổi chiều của speedX.

Trong lớp Ball sau, ta sẽ bổ sung thêm các thuộc tính right, bottom để tiện khi kiểm tra va chạm. Hai thuộc tính cx và cy là vị trí tâm của bóng và sẽ được khởi tạo ngẫy nhiên sao cho nó luôn nằm hoàn toàn bên trong canvas.

Ball.js:

```
function Ball(mapWidth, mapHeight){
        this.mapWidth = mapWidth;
        this.mapHeight = mapHeight;

        this.radius = 20;
        this.speedX = 3;
        this.speedY = 3;

        this.cx  =  Math.floor(Math.random()*(this.mapWidth-2*this.radius))  +
this.radius;
        this.cy  =  Math.floor(Math.random()*(this.mapHeight-2*this.radius))  +
this.radius;
```

```
}
Ball.prototype.draw = function(context){
        context.beginPath();
        context.fillStyle = "red";
        context.arc(this.cx,this.cy,this.radius,0,Math.PI*2,true);
        context.closePath();
        context.fill();
}
Ball.prototype.move = function(){
        this.cx += this.speedX;
        this.cy += this.speedY;

        this.left = this.cx - this.radius;
        this.top = this.cy - this.radius;
        this.right = this.cx + this.radius;
        this.bottom = this.cy + this.radius;
}
Ball.prototype.checkCollision = function(shapes) {

        if(this.left <= 0 || this.right >= this.mapWidth) this.speedX = -
this.speedX;
        if(this.top <= 0 || this.bottom >= this.mapHeight) this.speedY = -
this.speedY;

}
```

Sample.HTML:

```
<HTML>
<head>

<script src="ball.js"></script>
<script>

var _canvas;
var _context;
var _ball;

function draw(){
        _context.clearRect(0,0,_canvas.width,_canvas.height);
        _ball.draw(_context);
}
function update(){
        _ball.move();
        _ball.checkCollision();
        draw();
}

window.onload = function(){

        var interval = 10;
        _canvas = document.getElementById("canvas");
        _context = _canvas.getContext("2d");
        _ball = new Ball(_canvas.width,_canvas.height);

        setInterval("update()",interval);
}
</script>
</head>
```

```
<body>
   <canvas id="canvas" width="400px" height="300px" style="border: 1px solid
gray;"></canvas>
</body>
</HTML>
```

2. Thêm hiệu ứng bóng di chuyển

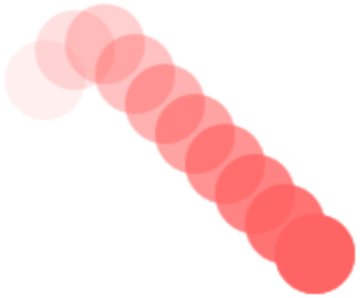
Để ví dụ không quá đơn điệu, ta sẽ thêm các ảnh bóng mờ di chuyển phía sau trái bóng chính. Ta sẽ lưu các vị trí bóng di chuyển qua vào một mảng có tối đa 10 phần tử và vẽ ra canvas trong hàm draw(). Phương thức Ball.draw() cần chỉnh sửa một chút để cho phép nhận tham số alpha xác định độ mờ đục. Tham số alpha này có giá trị nằm giữa đoạn 0 và 1:

```
Ball.prototype.draw = function(context,alpha){
        if(!alpha)
                alpha = 255;
        context.beginPath();
        context.fillStyle = "rgba(255, 100, 100," + alpha + ")";
        context.arc(this.cx,this.cy,this.radius,0,Math.PI*2,true);
        context.closePath();
        context.fill();

}
```

Để các trái bóng không nằm quá sát nhau, ta chỉ thực hiện lưu vị trí của bóng sau mỗi 5 lần bóng di chuyển hay sau mỗi 5 lần phương thức update() được gọi. Phương thức traceBall() sau sẽ tạo ra một đối tượng Ball mới với hai giá trị cx và cy lấy từ trái bóng chính và lưu vào mảng _balls. Khi chiều dài của mảng _balls vượt quá 10, ta dùng phương thức Array.splice() để cắt đi phần tử nằm ở đầu mảng:

```
function traceBall(ball){
        var b = new Ball;
        b.cx = ball.cx;
        b.cy = ball.cy;

        _balls.push(b);
        if(_balls.length>10)
                _balls.splice(0,1);
}
```

Kết quả:



3. Kiểm tra va chạm

Phần quan trọng của ví dụ này nằm ở phương thức kiểm tra va chạm của trái bóng. Phương thức này sẽ duyệt qua tất cả các chướng ngại vật và kiểm tra khoảng cách của nó so với chướng ngại vật theo hai phương dọc và ngang.

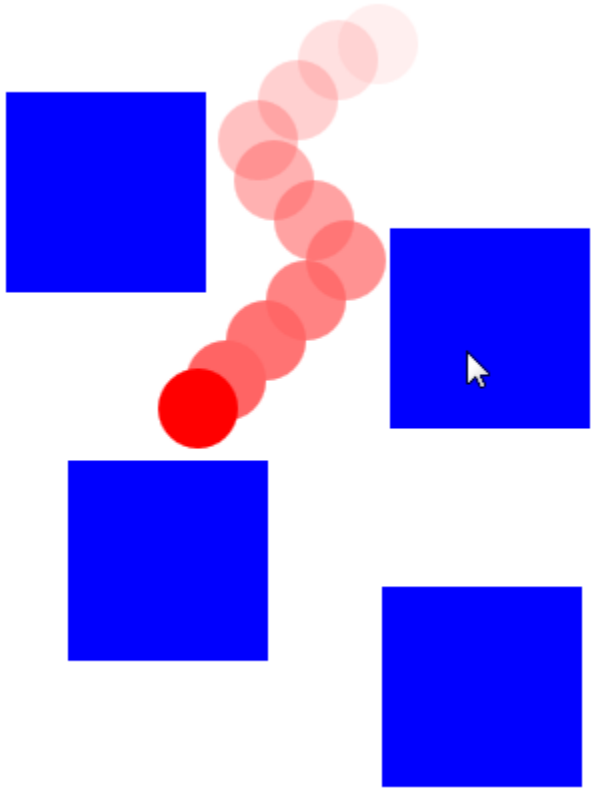
Với ví dụ này, ta kiểm tra va chạm ở mức rất đơn giản nên việc phản xạ của trái bóng chưa hoàn toàn chính xác. Ta sẽ giới thiệu các kĩ thuật kiểm tra va chạm và di chuyển trong các bài viết sau.

Phương thức kiểm tra va chạm của lớp Ball:

```
Ball.prototype.checkCollision = function(shapes) {

	if(this.left <= 0 || this.right >= this.mapWidth) this.speedX = -
this.speedX;
	if(this.top <= 0 || this.bottom >= this.mapHeight) this.speedY = -
this.speedY;

	for(var i = 0; i < shapes.items.length ; i++){

		var item = shapes.items[i];
		var hCollision = false;
		var vCollision = false;

		if(	this.right >= item.left && this.left <= item.right &&		// the
			((this.cy < item.top && this.bottom >= item.top) ||		// on
			(this.cy > item.bottom && this.top <= item.bottom)))
	// under the rectangle
		{
			this.speedY = -this.speedY;
			vCollision = true;
		}

		if(this.bottom >= item.top && this.top <= item.bottom &&
	// the ball is in the
			((this.cx < item.left && this.right >= item.left) ||
	// left or
			(this.cx > item.right && this.left <= item.right)))		// rig
		{
			this.speedX = -this.speedX;
			hCollision = true;
		}
```

```
                if(hCollision || vCollision)
                        break;
        }

}
```

Minh họa:



D. Kĩ thuật lập trình Game – Cơ bản

I. Vòng lặp game (Game loop) hoạt động thế nào?

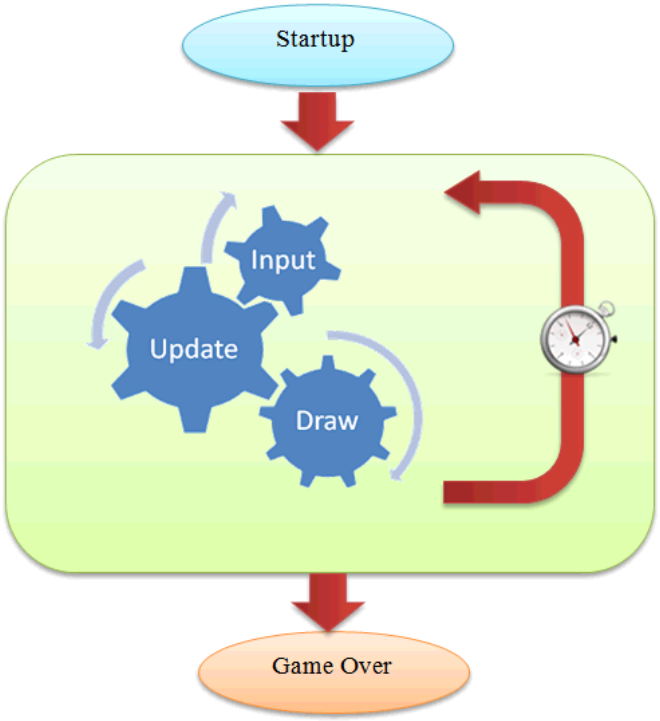
Phần cốt lõi của hầu hết các game chính là vòng lặp được dùng để cập nhật và hiển thị trạng thái của game. Trong bài viết này, ta sẽ minh họa các phương pháp tạo vòng lặp game với ngôn ngữ javascript.

1. Vòng lặp cơ bản

Một vòng lặp game cơ bản bao gồm các việc được thực hiện theo thứ tự sau:

```
while(gameRunning)
{
        processInput(); // keyboard, mouse,...
        updateGame();
        draw();
        // checkGameOver();
}
```

Minh họa:



Trong javascript, thay vì dùng vòng lặp, ta sẽ thay thế bằng setInterval() hoặc setTimeout(). Thông thường bạn chỉ cần xác định một giá trị interval thích hợp theo tốc độ của game.

```
function gameLoop()
{
        processInput();
        updateGame();
        draw();
        setTimeout(gameLoop,100); // 10 fps
}
```

Hợp lý hơn khi bạn muốn xác định rõ số khung hình/giây (fps):

```
const FPS = 60;
function gameLoop()
{
        // …
}
window.setInterval(gameLoop,1000/FPS);
```

2. Vòng lặp có tính toán thời gian

Tuy nhiên, không phải lúc nào công việc update và draw cũng hoàn thành trước khi lần gọi kế tiếp được thực hiện. Như vậy tốc độ của game sẽ không đồng nhất trên các thiết bị có cấu hình khác nhau.

Để giải quyết, ta sẽ sử dụng đến thời gian hệ thống để so sánh và quyết định thời điểm update/draw.

```
const FPS = 60;
const TICKS = 1000/FPS;
var lastUpdateTime;

function gameLoop()
{
        var now = Date.now();
        var diffTime = now - lastUpdateTime;
        if(diffTime >= TICKS)
                processInput();
                updateGame();
                lastUpdateTime = now;
        }
        draw();
        var sleepTime = TICKS - diffTime;
        if(sleepTime<0)
                sleepTime = 0;
        setTimeout(gameLoop,sleepTime);
}
```

Phương pháp trên chạy ổn với giá trị diffTime nhỏ hơn TICKS. Nghĩa là tốc độ của game không vượt quá giá trị TICKS cho phép. Tuy nhiên trong trường hợp diffTime lớn, việc cập nhật sẽ diễn ra chậm.

3. Giải pháp cuối cùng

Giải pháp của ta là sẽ thực hiện update với số lần dựa vào tỉ lệ diffTime/TICKS trong một lần lặp của game. Sẽ hiệu quả hơn nếu ta bỏ qua việc draw và thực hiện update liên tiếp vì sẽ giúp tăng tốc độ game để bù vào khoảng thời gian bị trì hoãn.

```
const FPS = 60;
const TICKS = 1000/FPS;

var lastUpdateTime;

function gameLoop()
{
        var diffTime = Date.now() - lastUpdateTime;
        var numOfUpdate = Math.floor(diffTime/TICKS);
        for(var i = 0;i < numOfUpdate;i++){
                processInput();
                updateGame();
        }
        if(diffTime >= TICKS)
                draw();

        lastUpdateTime += TICKS * numOfUpdate;
        diffTime -= TICKS * numOfUpdate;
        var sleepTime = TICKS - diffTime;
        setTimeout(gameLoop,sleepTime);
}
```

Nếu bạn sử dụng requestAnimationFrame cho vòng lặp game, bạn sẽ không cần quan tâm đến việc tính toán giá trị sleepTime.

4. Ví dụ hoàn chỉnh

Kiểm tra ví dụ này và so sánh với các phương pháp trước đó, thực hiện một vài công việc "quá tải" nào đó trên trình duyệt và bạn sẽ thấy khác biệt:

```
const FPS = 6;
const TICKS = 1000 / FPS;
var startTime;
var expectedCounter = 0;
var lastUpdateTime;
var actualCounter = 0;
var output;

function gameLoop()
{
    var diffTime = Date.now() - lastUpdateTime;
    var numOfUpdate = Math.floor(diffTime/TICKS);
    for(var i = 0;i < numOfUpdate;i++){
        updateGame();

    }
    if(diffTime >= TICKS)
```

```
        draw();

    lastUpdateTime += TICKS * numOfUpdate;
    diffTime -= TICKS * numOfUpdate;
    var sleepTime = TICKS - diffTime;
    setTimeout(gameLoop,sleepTime);
}

function updateGame() {
    actualCounter++;
}

function draw() {
    var s = "Actual updates: "+actualCounter;
    s += "<br/>Expected updates: "+Math.floor((Date.now()-startTime)/TICKS);
    output.innerHTML = s;
}
// onLoad
output = document.getElementById("output");
startTime = Date.now();
lastUpdateTime = startTime;
gameLoop();
```

Output:

Actual updates: 1323
Expected updates: 1323

II. Kiểm tra va chạm: hình tròn và chữ nhật

Thông thường các đối tượng trong game sẽ được xác định va chạm bằng cách đưa chúng về một dạng hình học cơ bản như hình chữ nhật, hình tròn. Bài viết này sẽ giúp bạn cách tính toán để kiểm tra va chạm giữa hai loại hình học này.

1. Giữa hai hình chữ nhật

Phương pháp: kiểm tra từng đỉnh của hình này có nằm bên trong hình kia không.

```
Rect.prototype.collideWidthRect = function(rect) {
        return this.contains(rect.left,rect.top) ||
                this.contains(rect.right,rect.top) ||
                this.contains(rect.right,rect.bottom) ||
                this.contains(rect.left,rect.bottom);
}
```

Cách trên không phải cách nhanh nhất, vì vậy bạn có thể dùng cách sau, đơn giản và hiệu quả hơn:

```
Rect.prototype.collideWidthRect = function(rect) {
        return !(this.left > rect.right ||
```

```
                        this.right < rect.left ||
                        this.top > rect.bottom ||
                        this.bottom < rect.top);
}
```

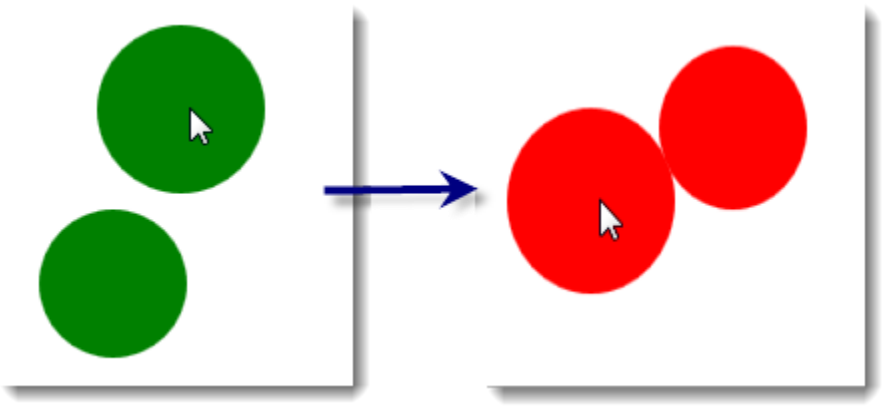
2. Giữa hai hình tròn

Phương pháp: Bởi vì mọi điểm nằm trên đường tròn cách đều tâm, nên việc kiểm tra va chạm giữa hai hình tròn sẽ được xác định dựa vào khoảng cách tâm giữa chúng. Để xác định khoảng cách giữa hai điểm, ta dựa vào định lý Pythagoras (Pythagorean theorem) . Ta coi khoảng cách giữa hai điểm là đường chéo của một tam giác vuông. Vậy độ lớn của đường chéo này là:

```
c² = a² + b²
=> c = sqrt(a² + b²)
```

```
Circle.prototype.collideWithCircle = function(circle){
        var dx = this.cx - circle.cx;
        var dy = this.cy - circle.cy;

        return Math.sqrt(dx*dx + dy*dy) <= this.radius+circle.radius;
}
```

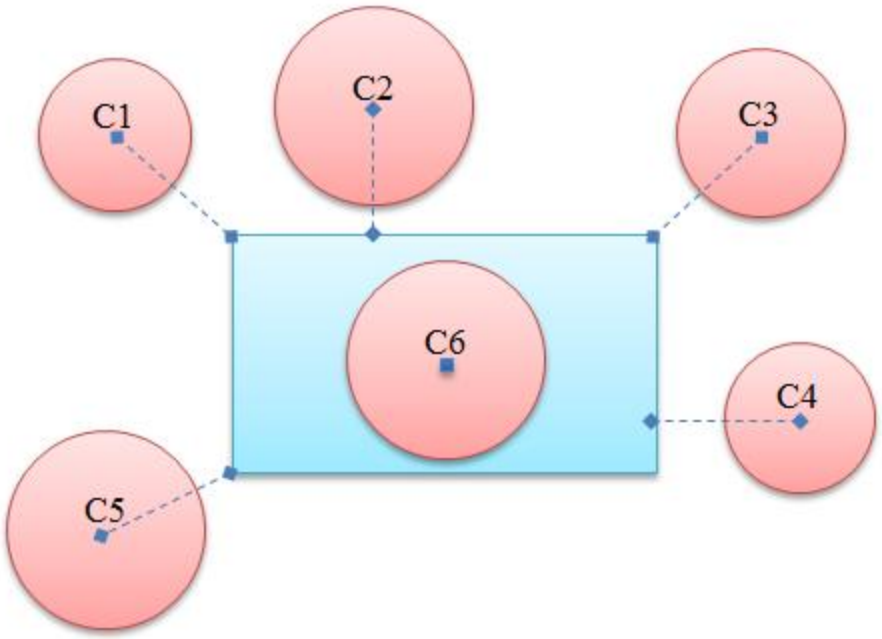
Trong minh họa dưới đây, hai hình tròn có màu mặc định là xanh lá, khi va chạm nhau, chúng sẽ chuyển sang màu đỏ.



3. Giữa hình tròn và hình chữ nhật

Phương pháp: Gọi C là tâm và R là bán kính hình tròn. Ta sẽ tìm cách xác định điểm A là điểm gần nhất thuộc hình chữ nhật đến tâm C. Sau đó so sánh độ lớn của CA với R.

Khoảng cách giữa tâm C hình tròn và điểm A của hình chữ nhật được minh họa như hình dưới đây. Khi tâm hình tròn nằm bên trong hình chữ nhật, thì điểm C và A sẽ trùng nhau.



Gọi rect là hình chữ nhật cần xác định va chạm. Ta có thuật toán để xác định điểm A như sau:

- B1: Gán A bằng với C.
- B2: Nếu C.X < rect.Left, đặt A.X = rect.Left. Ngược lại nếu C.X > rect.Right, đặt A.X = rect.Right.
- B3: Nếu C.Y < rect.Top, đặt A.Y = rect.Top. Ngược lại nếu C.Y > rect.Bottom, đặt A.Y = rect.Bottom.

Khi đã có điểm A, ta lại dùng công thức Pythagoras để so sánh với bán kính của hình tròn.

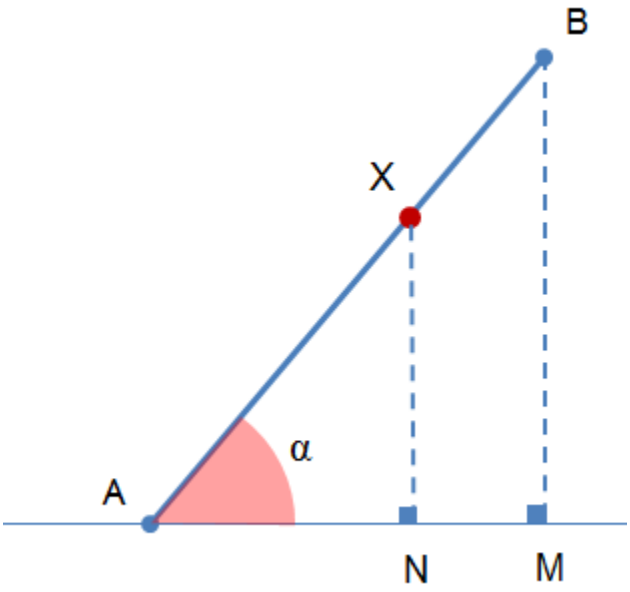
```
Circle.prototype.collideWithRect = function(rect){
        var px = this.cx;
        var py = this.cy;

        if(px < rect.left)
                px = rect.left;
        else if(px > rect.right)
                px = rect.right;

        if(py < rect.top)
                py = rect.top;
        else if(py > rect.bottom)
                py = rect.bottom;

        var dx = this.cx - px;
        var dy = this.cy - py;

        return (dx*dx + dy*dy) <= this.radius*this.radius;
}
```

III. Kiểm tra một điểm nằm trên đoạn thẳng

Có nhiều cách để kiểm tra một điểm có thuộc đường thẳng (tương tự với đoạn thẳng) hay không: bằng cách sử dụng các công thức hình học hoặc thuật toán vẽ đường thẳng,… Ngoài những cách trên, ta sẽ giới thiệu một phương pháp đơn giản nhất là sử dụng phép so sánh góc để giải quyết vấn đề này.

Trong demo sau, khi bạn di chuyển chuột ngang lên trên đoạn thẳng, bạn sẽ đoạn thẳng chuyển sang màu đỏ.

Ý tưởng:
Ta có đoạn thẳng AB tạo nên góc α (so với phương ngang hoặc phương bất kì), và mọi điểm thuộc AB đều tạo nên góc α so với phương ngang.
Hay nói cách khác, ta xem đoạn thẳng AB là đường chéo của một tam giác vuông ABM. Tỉ lệ giữa hai cạnh góc vuông của tam giác này sẽ bằng với tỉ lệ hai cạnh góc vuông tạo bởi tam giác vuông AXN. Minh họa như hình dưới đây:



Như vậy việc xác định một điểm X có thuộc AB hay không chỉ cần dựa vào 2 yếu tố:

(1) X phải nằm trong vùng hình chữ nhật được tạo bởi đường chéo AB (ngoại tiếp tam giác ABM).

(2) Góc XAN và BAM phải bằng nhau.

Ngoài ra, do việc so sánh là kiểu số thực và có thể do độ rộng của đoạn thẳng khác nhau nên ta cần chấp nhận một giá trị sai số EPSILON nào đó.

Muốn chính xác hơn khi độ rộng của đoạn thẳng thay đổi, bạn có thể tính góc sai số cho phép bằng cách dựa vào ½ độ rộng đoạn thẳng và khoảng cách AX để tính. Tuy nhiên, điều này không quan trọng lắm và làm cho việc kiểm tra tốn thêm chi phí.

Để đơn giản, ta sẽ tính tỉ lệ (tan) thay vì tính góc α:

```
Line.prototype.contains = function(x, y) {
        if(     x < Math.min(this.handler1.cx,this.handler2.cx) ||
                x > Math.max(this.handler1.cx,this.handler2.cx) ||
                y < Math.min(this.handler1.cy,this.handler2.cy) ||
                y > Math.max(this.handler1.cy,this.handler2.cy))
                return false;
        var dx = this.handler1.cx-this.handler2.cx;
        var dy = this.handler1.cy-this.handler2.cy;
        var tan1 = Math.abs(dy/dx);
        var tan2 = Math.abs((y-this.handler1.cy)/(x-this.handler1.cx));
        return Math.abs(tan1 - tan2) < EPSILON;
}
```

IV. Vector 2D cơ bản

Ứng dụng của vector rất quan trọng trong lĩnh vực lập trình game và đồ họa. Thông qua vector, ta có thể mô phỏng được các chuyển động, tính toán lực, hướng di chuyển sau khi va chạm,…

1. Khái niệm

"*...một vectơ là một phần tử trong một không gian vectơ, được xác định bởi ba yếu tố: điểm đầu (hay điểm gốc), hướng (gồm phương và chiều) và độ lớn (hay độ dài).*" (Wikipedia)

Từ một đoạn thẳng AB ta có thể xác định được vector (mã giả):

u.root = A;
u.x = B.x-A.x;
u.y = B.y-A.y;
u.length = sqrt(u.x*u.x+u.y*u.y); // |u|

2. Vector đơn vị (Unit Vector, Normalized Vector)

Vector đơn vị của u là vector có chiều dài bằng 1 và kí hiệu là û. Vector u sau được gọi là vector đơn vị của v bằng cách tính (mã giả):

û = u/|u|

Như vậy:
|û| = 1

3. Tích vô hướng (Dot product, Scalar product)

Phép toán tích vô hướng của hai vector được biểu diễn dấu chấm (nên được gọi dot product) và được tính như sau:

v.u = |v||u|cosθ
= v1u1 + v2u2 + … + vnun

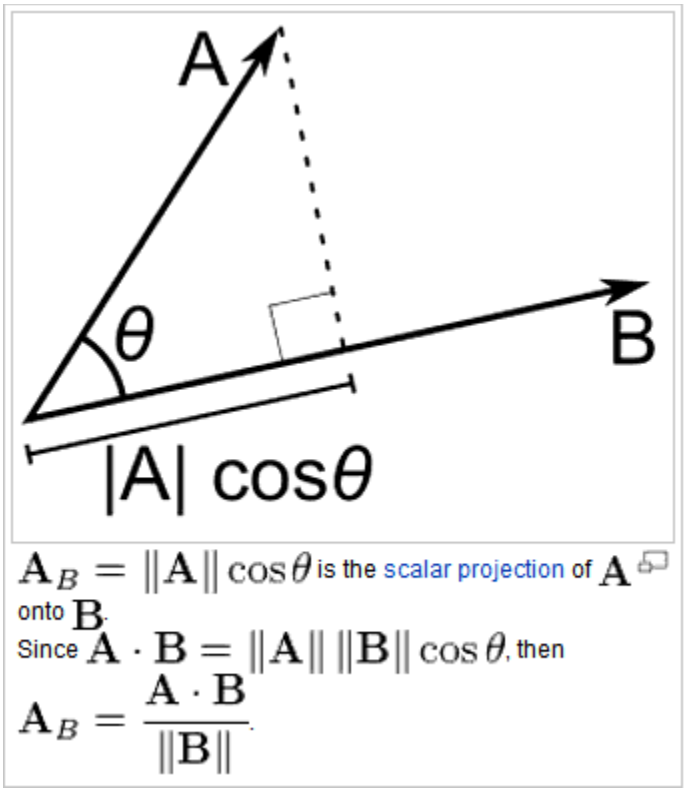
Với θ (theta) là góc giữa v, u và n là chiều của không gian.

Từ công thức trên, ta có thể tính được θ:

cosθ = (v.u)/(|v||u|)
θ = arccos(cosθ)

4. Phép chiếu (Projection)

Một ứng dụng khác của tích vô hướng là tính phép chiếu của một vector này lên vector khác. Tương tự như việc chiếu một vector v thành 2 giá trị x và y lên trục Oxy. (Hình từ wikipedia)



Một vector a được gọi là vector chiếu của v lên u nếu như nó có cùng phương với u và có độ lớn:

|a| = |v|cosθ

Tổng quát hơn với hai vector bất kì, xét biểu thức:
$|a| = |v|\cos\theta$

Với u là vector tạo với v một góc θ, từ công thức:

$\cos\theta = (v.u)/(|v||u|)$

Suy ra:

$|a| = |v|(v.u)/(|v||u|) = (v.u)/|v|$

Vậy ta đã tính được độ lớn của vector v chiếu trên vector u. Từ giá trị |x| này, ta có thể tính được vector x bằng cách nhân với vector đơn vị của u:

$a = |a|\hat{u}$

5. Hiện thực với javascript

Một đối tượng Line được biểu diễn bởi 2 điểm đầu (p1) và cuối (p2) của một đoạn thẳng. Từ hai điểm này, ta có phương thức getVector() để lấy về một đối tượng vector của đoạn thẳng.

```
Line.prototype.getVector = function() {
        var x = this.p2.x-this.p1.x;
        var y = this.p2.y-this.p1.y;

        return {
                x: x,
                y: y,
                root: this.p1,
                length: Math.sqrt(x*x+y*y)
        };
}
```

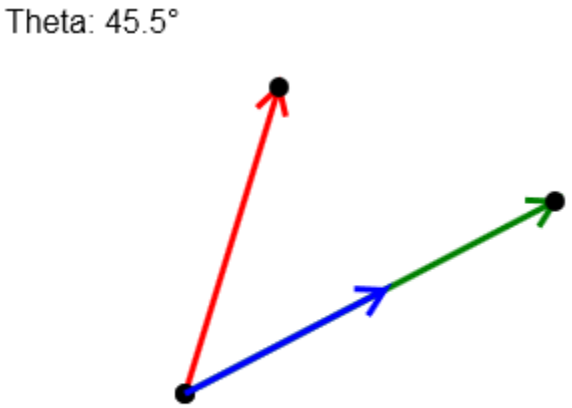
Phương thức update() sau được gọi mỗi khi một đoạn thẳng bị thay đổi:

```
function update(){
        var v1 = _line1.getVector();

        var v2 = _line2.getVector();

        // normalize vector v2
        v2.dx = v2.x/v2.length;
        v2.dy = v2.y/v2.length;

        // dot product
        var dp = v1.x*v2.x + v1.y*v2.y;
        // length of the projection of v1 on v2
        var length = dp/v2.length;

        // the projection vector of v1 on v2
        var v3 = {
```

```
                x: length*v2.dx,
                y: length*v2.dy,
        };
        // the projection line
        _line3.p2.x = _line3.p1.x+v3.x;
        _line3.p2.y = _line3.p1.y+v3.y;

        // calculate the angle between v1 and v2
        // and convert it from radians into degrees
        _angle = Math.acos(dp/(v1.length*v2.length))*(180/Math.PI);
        _angle = Math.round(_angle*100)/100;
}
```

Trong phần demo sau, ta thực hiện phép chiếu vector màu đỏ lên vector xanh lá. Kết quả của phép chiếu là vector màu xanh lam. Bạn có thể thay đổi hướng và độ lớn của vector bằng cách nhấn nhấn và rê chuột vào đầu mũi tên.

Xem Demo.



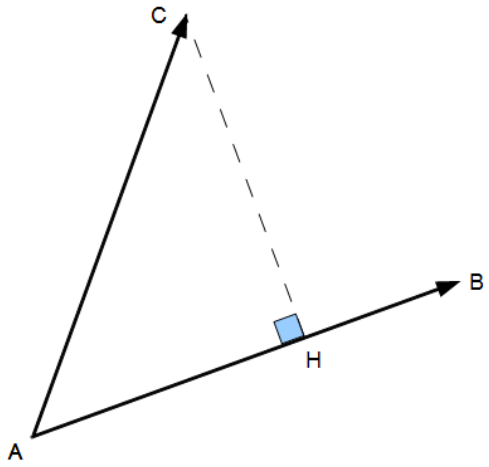
V. Khoảng cách từ điểm đến đoạn thẳng

Dựa vào phép chiếu từ tích vô hướng (Dot product) của hai vector, ta có thể tính được khoảng cách từ một điểm đến một đường thẳng, đoạn thẳng.

Bài toán: Tìm khoảng cách từ điểm C đến đoạn thẳng AB.
Giải quyết:
Ta sẽ tính vector chiếu của AC lên AB và gọi là AH. Vì CH vuông góc với AB và khoảng cách từ C đến AB chính là CH. Như vậy ta tính được khoảng cách từ một điểm đến một đường thẳng chứa AB.

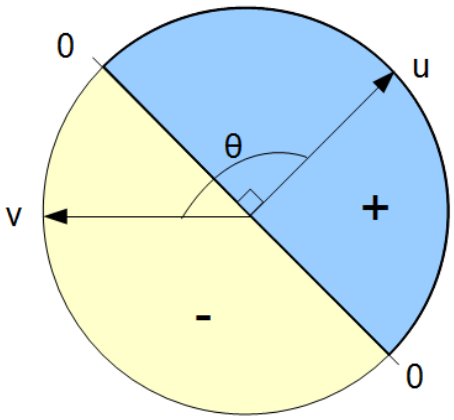
Để tìm khoảng cách đến đoạn thẳng, ta cần xác định H có thuộc AB hay không.

Ta xem lại một chút về tích vô hướng của hai vector.



Kết quả của tích vô hướng có 3 khoảng giá trị để ta có thể ước lượng được góc giữa hai vector (bỏ qua chiều âm/dương):

Case 1 (Dot product < 0): Góc giữa hai vector lớn hơn 90 độ. Điểm H nằm ngoài AB.
Case 2 (Dot product = 0): Hai vector vuông góc với nhau (90 độ). Điểm H trùng với A.
Case 3 (Dot product > 0): Góc giữa hai vector nhỏ hơn 90 độ. Điểm H có thể nằm trong AB hoặc không. Điểm H sẽ nằm ngoài AB nếu như độ dài AH > AB.



Range of the Dot Product

Mã nguồn:

```
function pointLineDistance(){

        // v1: AC
        // v2: AB
        // v3: AH

        var v1 = _line1.getVector();
        var v2 = _line2.getVector();
```

```
        // normalize vector v2
        v2.dx = v2.x/v2.length;
        v2.dy = v2.y/v2.length;

        // dot product
        var dp = v1.x*v2.x + v1.y*v2.y;
        // length of the projection of v1 on v2
        var length = dp/v2.length;

        // the projection vector of v1 on v2
        var v3 = {
                x: length*v2.dx,
                y: length*v2.dy,
        };
        v3.length = Math.sqrt(v3.x*v3.x+v3.y*v3.y);

        // the projection line
        _line3.p2.x = _line3.p1.x+v3.x;
        _line3.p2.y = _line3.p1.y+v3.y;

        var d; // distance
        // d = -1 means H does not lie on AB

        if(dp < 0)
        {
                d = -1;
        }
        else
        {
                if(v3.length > v2.length)
                        d = -1;
                else
                {
                        var dx = v1.x-v3.x;
                        var dy = v1.y-v3.y;
                        d = Math.sqrt(dx*dx+dy*dy);
                }
        }

        return d;
}
```

VI. Giao điểm của hai đường thẳng

Từ hai đoạn thẳng (hoặc vector) trong mặt phẳng 2D, ta có thể tìm được giao điểm của chúng để tính toán các góc phản xạ và hướng di chuyển.

1. Tạo phương trình đường thẳng từ đoạn thẳng

Ta có hai điểm A(x1,y1) và B(x2,y2) tạo thành một đoạn thẳng, để tạo được một phương trình đường thẳng từ hai điểm này, ta cần tính được độ nghiêng của đường thẳng (slope) theo công thức:

a = (y2 – y1)/(x2 – x1)

Thay thế x2, y2 bằng hai biến x,y:

a = (y – y1)/(x – x1)
=> y – y1 = a(x – x1)

Hay:
y = ax + (y1 – ax1)
Đặt b = y1 – ax1, ta có:

y = ax + b

2. Tính giao điểm của hai đường thẳng

Ta có hai phương trình đường thẳng

(1): y = a1x + b1
(2): y = a2x + b2

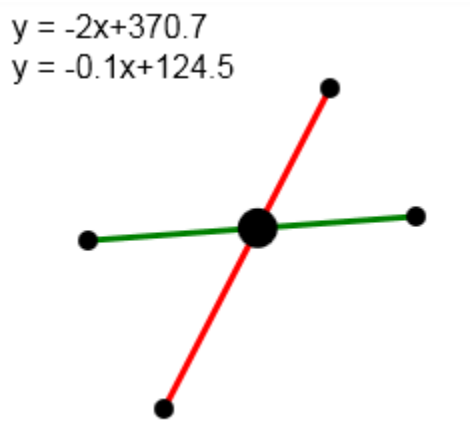
Ta có thể tính được giao điểm của chúng bằng cách tìm giao điểm x trước:

a1x + b1 = a2x + b2;

=> x(a1 – a2) = b2 – b1
=> x = (b2 – b1)/(a1 – a2)

Khi đã có x, ta sẽ thế vào (1) hoặc (2) để tính được y. Để kiểm tra hai đoạn thẳng có cắt nhau không, ta chỉ cần kiểm tra điểm {x,y} thuộc cả hai đoạn thẳng hay không. Ngoài ra ta cần loại trừ trường hợp hai đường thẳng song song hoặc trùng nhau, khi đó giá trị slope của chúng sẽ bằng nhau.

3. Minh họa với HTML5 Canvas



(Đoạn mã khá dài nên có thể xem trên trang yinyangit.wordpress.com).

VII. Va chạm và phản xạ

Để tính được hướng phản xạ của khi một vật thể va chạm vào mặt phẳng, ta có thể dựa vào góc nghiêng của mặt phẳng và vector theo từng vùng giá trị. Tuy nhiên cách tổng quát hơn là sử dụng các phép toán trên vector để thực hiện.

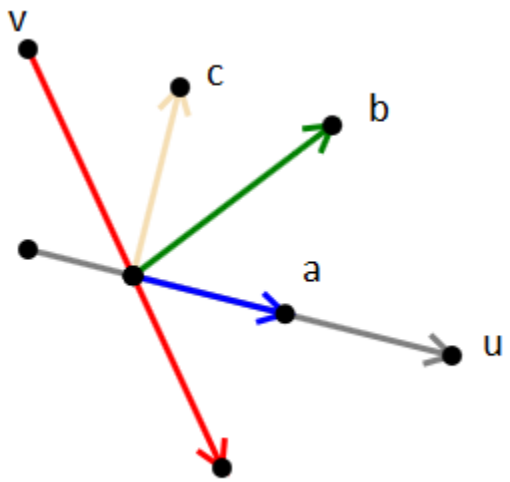
Xem Demo

1. Kiểm tra hai đoạn thẳng cắt nhau

Từ bài viết trước về tìm giao điểm của hai đường thẳng, ta có thể kiểm tra xem giao điểm tìm được có phải là giao điểm của hai đoạn thẳng hay không. Cách kiểm tra đơn giản là bạn xem điểm này có thuộc cả hai phần bao hình chữ nhật của hai đoạn thẳng. Ta có phương thức kiểm tra như sau:

```
// detect if a point is inside the rectangle bound of this line
Line.prototype.insideRectBound = function(x, y){
        if(     Math.min(this.p1.x,this.p2.x) - x > EPSILON    ||
                x - Math.max(this.p1.x,this.p2.x) > EPSILON ||
                Math.min(this.p1.y,this.p2.y) - y > EPSILON ||
                y - Math.max(this.p1.y,this.p2.y) > EPSILON)
                return false;
        return true;
}

// ...
var intersected = _line1.insideRectBound(x,y) && _line2.insideRectBound(x,y);
```

2. Phương pháp



Trong hình minh họa sau, vector v là hướng di chuyển của vật thể va chạm vào u. Vector phản xạ của v là b. Từ vector b ta phân tích ra hai vector thành phần là a và c (như vậy ta có a + c = b). Trong đó: a: vector chiếu của v trên u. c: vector chiếu của v trên đường thẳng vuông góc với u. Đây là vector pháp tuyến của u và có độ dài bằng khoảng cách từ gốc của v đến u. Như vậy để tìm được vector phản xạ b, ta chỉ cần theo 4 bước:

1. Tìm giao điểm của hai đoạn thẳng (tương ứng với hai vector v và u).

2. Tìm vector chiếu a của v trên u.

3. Tìm vector pháp tuyến của u có độ dài bằng khoảng cách từ gốc của v đến u.

4. Tính tổng vector a và c.

Xem mã nguồn minh họa bằng javascript tại trang sau để thấy chi tiết hơn:

http://yinyangit.wordpress.com/2012/04/10/gamedev-vector2d-va-cham-va-phan-xa/

VIII. Va chạm giữa đường tròn và đoạn thẳng

Tìm điểm va chạm của một đường tròn với đoạn thẳng và xác định hướng phản xạ của di chuyển thông qua các phép toán trên vector.

1. Va chạm

Để xác định va chạm, ta tính toán khoảng cách từ tâm đường tròn đến đoạn thẳng theo phương pháp Tính khoảng cách từ điểm đến đoạn thẳng và so sánh với bán kính của đường tròn.

Ví dụ trong phần kiểm tra khoảng cách này chưa xét trường hợp va chạm với hai đầu của đoạn thẳng. Bởi vì việc kiểm tra phản xạ khi va chạm với đầu đoạn thẳng sẽ cần tính toán thêm một vài bước nên ta sẽ bỏ qua.

Phương thức sau sẽ trả về điểm va chạm của trái bóng với đoạn thẳng hoặc null nếu như không có.

```
Ball.prototype.findCollisionPoint = function(line) {

        // create a vector from the point line.p1 to the center of this ball.
        var v1= {
                x: this.cx - line.p1.x,
                y: this.cy - line.p1.y
        };

        var v2 = line.getVector();

        var v3 = findProjection(v1,v2);

        v3.length = Math.sqrt(v3.x*v3.x+v3.y*v3.y);

        var collisionPoint = null;

        if(v3.dp>0 && v3.length <= v2.length)
        {
                var dx = v1.x-v3.x;
                var dy = v1.y-v3.y;
                var d = Math.sqrt(dx*dx+dy*dy);        // distance

                if(d>this.radius)
                        return null;

                collisionPoint = {
                        x: line.p1.x + v3.x,
                        y: line.p1.y + v3.y
                };
                // don't overlap
                if(d < this.radius)
                {
                        this.cx -=
        (this.movement.x/this.speed)*(this.radius-d);
                        this.cy -=
        (this.movement.y/this.speed)*(this.radius-d);
                }
        }
        return collisionPoint;
}
```

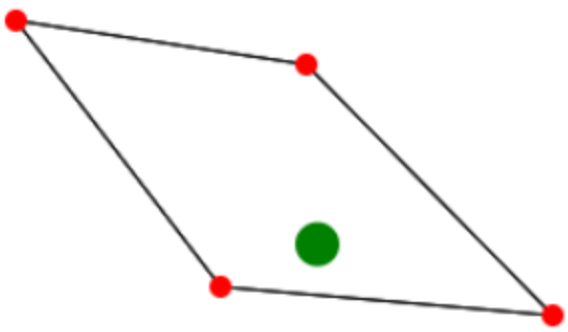
2. Phản xạ

Dựa vào vector chuyển động của trái bóng và vector được tạo ra từ đoạn thẳng, ta có thể tính được vector chuyển động mới của bóng (xem Vector2D: Va chạm và phản xạ).

```
Ball.prototype.bounceAgainst = function(line) {

        if(this.findCollisionPoint(line))
        {
                var v1 = this.movement;

                var v2 = line.getVector();

                var v3 = findProjection(v1,v2);

                // perpendicular vector of v2
                var v4 = {
                        x: v2.y,
                        y: -v2.x
                };

                v4 = findProjection(v1,v4);
                v4.x = -v4.x;
                v4.y = -v4.y;

                // bounce vector
                var v5 = {
                        x: v3.x + v4.x,
                        y: v3.y + v4.y
                };

                v5.length = Math.sqrt(v5.x*v5.x+v5.y*v5.y);

                // normalize vector
                v5 = {
                        x: v5.x/v5.length,
                        y: v5.y/v5.length
                };
                this.setMovement(v5);
        }
}
```

Đối với việc kiểm tra phản xạ khi trái bóng va chạm với 1 điểm (như hai đầu của đoạn thẳng), bạn cần xác định một vector vuông góc với đường thẳng nối tâm trái bóng đến điểm va chạm và coi đó là mặt phẳng va chạm.

Minh họa:



IX. Va chạm giữa nhiều đường tròn

Để xác định hướng di chuyển của hai quả cầu có khối lượng khác nhau sau khi va chạm, ta sử dụng công thức của định luật bảo toàn động lượng trong một hệ cô lập.

Xét trường hợp va chạm trong không gian một chiều (1D – các vật di chuyển trên một đường thẳng), ta gọi m là khối lượng và vận tốc của quả cầu C, u là vận tốc trước khi va chạm và v là vận tốc sau khi va chạm. Tại thời điểm hai quả cầu C1 và C2 va chạm nhau, ta có công thức:

m1u1 + m2u2 = m1v1 + m2v2

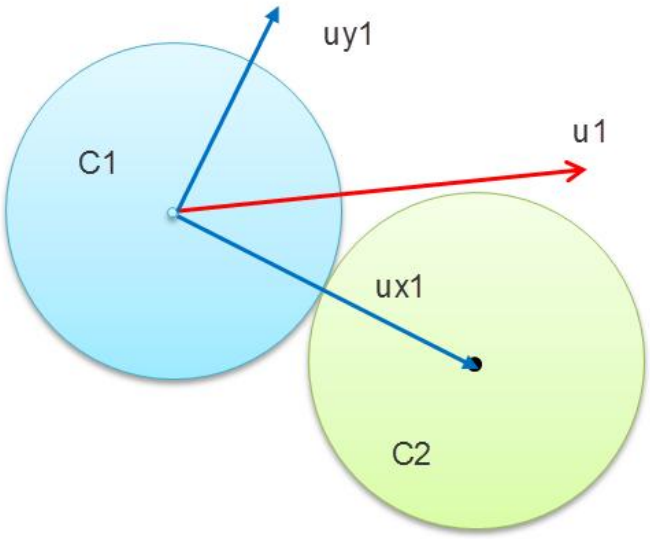
Suy ra:

v1 = (u1*(m1-m2) + 2*m2*u2)/(m1+m2)

v2 = (u2*(m2-m1) + 2*m1*u1)/(m1+m2)

Ta có thể áp dụng công thức này trong không gian hai chiều (2D) để tính vận tốc theo một phương xác định.

Bằng cách chia vector vận tốc thành hai vector thành phần, một vector ux1 nằm trên đường thẳng chứa hai tâm C1 và C2 (phương ngang), vector uy1 còn lại vuông góc với vx1 (phương dọc).ta chuyển không gian 2 chiều thành 1 chiều theo phương chứa đoạn thẳng nối hai tâm đường tròn.




```
var angle1 = Math.atan2(u1.y, u1.x);
var ux1 = u1.length * Math.cos(angle1-angle);
var uy1 = u1.length * Math.sin(angle1-angle);
```

Áp dụng công thức bên trên, ta tính được vector vận tốc mới theo phương ngang là vx1. Do đây là không gian 1D nên vận tốc uy1 sẽ không ảnh hưởng.

```
var vx1 = ((ball1.mass-ball2.mass)*ux1+(ball2.mass+ball2.mass)*ux2)/(ball1.mass+ball2.mass);
var vy1 = uy1;
```

Đã có hai vector thành phần là vx1 và vy1, ta cần chuyển lại các vector mới này sang không gian 2D và hợp thành vector vector chuyển động mới là v1. Do vector vx1 vuông góc với vy1 nên ta cần cộng với angle một góc 90 độ hay PI/2. Ở đây, angle là góc tạo bởi đoạn thẳng nối hai tâm đường tròn.

```
var v1 = {};
v1.x = Math.cos(angle)*vx1+Math.cos(angle+Math.PI/2)*vy1;
v1.y = Math.sin(angle)*vx1+Math.sin(angle+Math.PI/2)*vy1;
```

1. Xử lý va chạm của nhiều đường tròn

Cách đơn giản nhất và có độ phức tạp tương đối nhỏ (số lần lặp là n(n-1)/2) trong việc kiểm tra và xử lý va chạm cho nhiều đối tượng là sử dụng hai vòng lặp lồng nhau theo dạng sau:

```
for(var i=0;i<_balls.length;i++)
{
        for(var j=i+1;j<_balls.length;j++)
                checkCollision(_balls[i],_balls[j]);
}
```

Hàm checkCollision:

```
function checkCollision(ball1,ball2){

        var dx = ball1.cx - ball2.cx;
        var dy = ball1.cy - ball2.cy;

        // check collision between two balls
        var distance =  Math.sqrt(dx*dx+dy*dy);
        if(distance > ball1.radius + ball2.radius)
                return;

        var angle = Math.atan2(dy,dx);

        var u1 = ball1.getVelocity();
        var u2 = ball2.getVelocity();

        var angle1 = Math.atan2(u1.y, u1.x);
        var angle2 = Math.atan2(u2.y, u2.x);
        var ux1 = u1.length * Math.cos(angle1-angle);
        var uy1 = u1.length * Math.sin(angle1-angle);
        var ux2 = u2.length * Math.cos(angle2-angle);
        var uy2 = u2.length * Math.sin(angle2-angle);

        var                 vx1                 =                 ((ball1.mass-
ball2.mass)*ux1+(ball2.mass+ball2.mass)*ux2)/(ball1.mass+ball2.mass);
        var        vx2        =        ((ball1.mass+ball1.mass)*ux1+(ball2.mass-
ball1.mass)*ux2)/(ball1.mass+ball2.mass);
        var vy1 = uy1;
        var vy2 = uy2;

        // now we transform the 1D coordinate back to 2D
        var v1 = {}, v2 = {};
        v1.x = Math.cos(angle)*vx1+Math.cos(angle+Math.PI/2)*vy1;
        v1.y = Math.sin(angle)*vx1+Math.sin(angle+Math.PI/2)*vy1;

        v2.x = Math.cos(angle)*vx2+Math.cos(angle+Math.PI/2)*vy2;
        v2.y = Math.sin(angle)*vx2+Math.sin(angle+Math.PI/2)*vy2;

        ball1.velocity = v1;
        ball2.velocity = v2;
}
```

X. Kiểm tra va chạm dựa trên pixel

Các đối tượng đồ họa (hoặc hình ảnh) trong game thường được một giới hạn trong một khung bao hình chữ nhật có nền trong suốt (pixel có alpha = 0). Như vậy đối với các đối tượng phức tạp và muốn kiểm tra va chạm chính xác, ta cần kiểm tra các pixel có độ alpha > 0 của hai đối tượng đồ họa có cùng nằm trên một vị trí hay không.

1. Một wrapper của Image

Để tiện xử lý các hình ảnh trong canvas dưới dạng một đối tượng trong game với các chức năng cần thiết, ta tạo một lớp ImageObj chứa bên trong một đối tượng Image để lưu trữ hình ảnh. Phương thức quan trọng mà ImageObj phải có là draw(), tuy nhiên việc nạp ảnh có thể diễn ra khá lâu vì vậy ta cần một cách thức xử lý trường hợp này. Chẳng hạn ta sẽ viết ra một dòng thông báo thay thế cho ảnh với kích thước mặc định trong trường hợp ảnh chưa được tải xong:

```
function ImageObj(){
	// ...
	this.draw = function(context){
		if(ready){
			context.drawImage(this.img,this.left,this.top);
		}
		else{
			// image has not finished loading
			// draw something useful instead
			context.save();
			context.fillText("Image is not
ready",this.left+10,this.top+10);
			context.restore();
		}
		context.strokeRect(this.left,this.top,this.width,this.height);
		context.stroke();
	};
	// ...
}
```

Chưa đủ, để đối tượng ImageObj có thể tự vẽ lại sau ảnh được nạp xong, đồng thời lấy được đối tượng ImageData (chứa mảng các pixel), ta sẽ viết một số lệnh xử lý trong sự kiện onload của Image.

```
function ImageObj(){
	// ...
	var self = this;
	this.img.onload = function(){
		self.width = this.width;
		self.height = this.height;
		ready = true; // this image is ready to use

		// draw image after loading
		context.clearRect(self.left,self.top,self.width,self.height);
		self.draw(context);
		// get ImageData from this image
		self.data =
context.getImageData(self.left,self.top,self.width,self.height).data;
	};
	this.img.src = url;
}
```

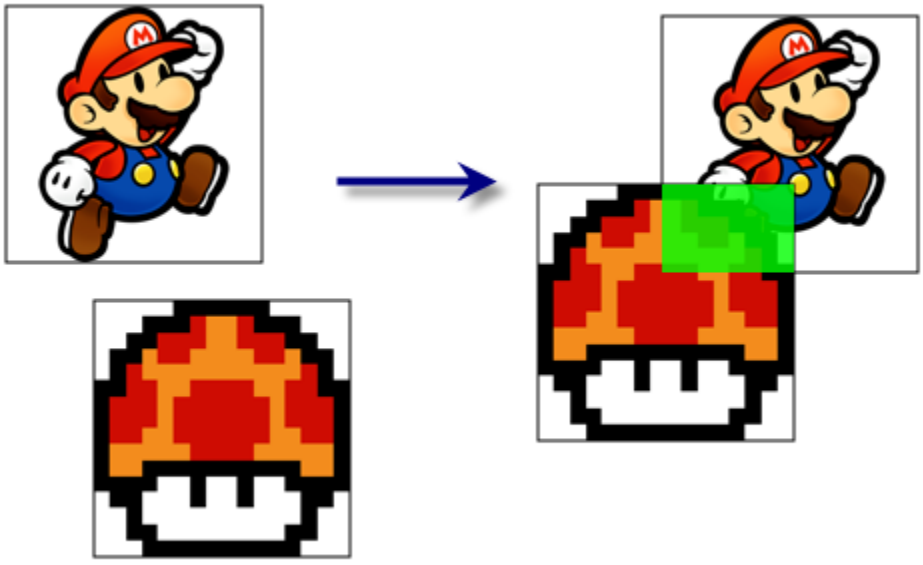
Do vấn đề bảo mật, ta sẽ không lấy được ImageData của các ảnh không nằm trong host hiện tại (khác host mà script được thực thi).

2. Xác định vùng giao hai hình chữ nhật

Thay vì lặp qua toàn bộ hai hình ảnh và kiểm tra từng pixel của chúng. Ta sẽ giới hạn lại phần ảnh cần kiểm tra bằng cách xác định vùng giao giữa hai hình ảnh. Phần này cũng giúp ta xác định nhanh hai đối tượng có thể xảy ra va chạm hay không.

```
function findIntersectionRect(img1,img2){

        var rect = {
                left: Math.max(img1.left,img2.left),
                top: Math.max(img1.top,img2.top),
                right: Math.min(img1.left+img1.width,img2.left+img1.width),
                bottom: Math.min(img1.top+img1.height,img2.top+img2.height)
        };

        if(rect.left>rect.right || rect.top>rect.bottom)
                return  null;
        return rect;
}
```

Xem Demo.

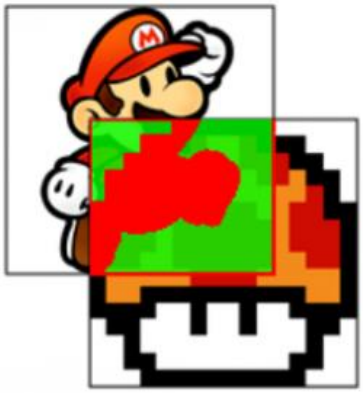


3. Kiểm tra va chạm

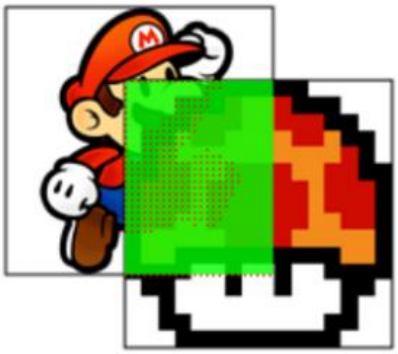
Nếu đã tìm hiểu về đối tượng ImageData, bạn biết rằng các pixel sẽ được lưu trữ trong một mảng một chiều. Mỗi pixel bao gồm bốn phần tử đứng liền nhau trong mảng theo thứ tự là [Red, Green, Blue, Alpha]. Như vậy một ảnh có kích thước 20×30 sẽ có 600 pixel và có 600×4=2400 phần tử trong ImageData.

```
function checkPixelCollision(img1,img2){
        if(!img1.data || !img2.data)
                return null;
        var rect =      findIntersectionRect(img1,img2);
        if(rect)
        {
                // this array will hold all collision points of two images
                var points = [];
                for(var i=rect.left;i<=rect.right;i++){
                        for(var j=rect.top;j<=rect.bottom;j++){
                                var index1 = ((i-img1.left)+(j-
img1.top)*img1.width)*4+3;
                                var index2 = ((i-img2.left)+(j-
img2.top)*img2.width)*4+3;
                                if(img1.data[index1+3]!=0 &&
img2.data[index2+3]!=0)
                                {
                                        points.push({x:i,y:j});
                                        //   you   can   exit   here   instead   of
continue looping
                                }
                        }
                }

                return {
                        rect: rect,
                        points: points
                };
        }
        return null;
}
```

Để tăng tốc độ kiểm tra va chạm, thay vì lặp qua từng pixel, bạn có thể lặp ngắt quãng n pixel (theo cả 2 chiều ngang và dọc) tùy theo mức độ yêu cầu chính xác của ứng dụng. Như vậy số lần lặp để kiểm tra sẽ giảm đi 2n lần tương ứng. Dưới đây là ảnh minh họa việc thay đổi bước tăng của vòng lặp từ 1 lên 3. Vùng màu đỏ là các pixel có độ alpha > 0 thuộc cả hai ảnh:



Step = 1



Step = 3

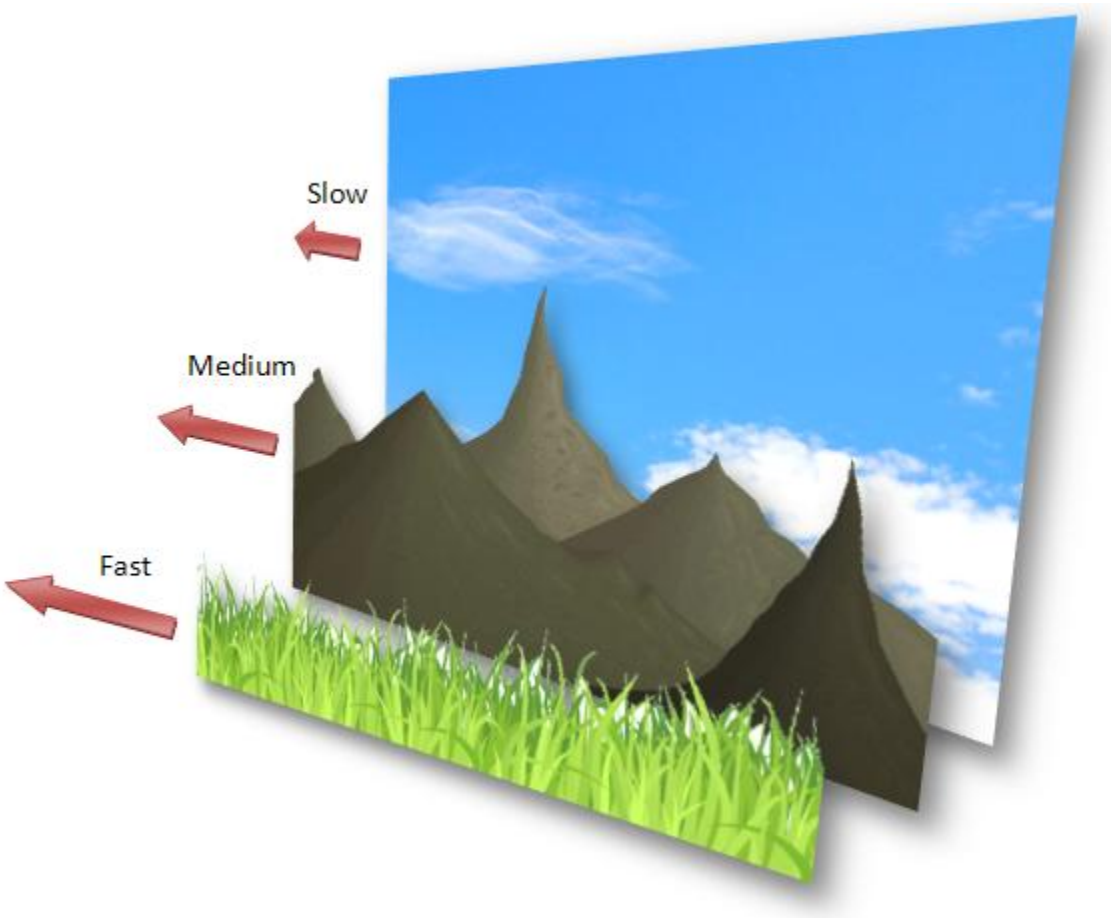
E. Kỹ thuật lập trình Game – Nâng cao

I. Cuộn ảnh nền và bản đồ (Map Scrolling)

Cuộn bản đồ là chức năng không thể thiếu trong những game có bản đồ lớn vượt quá kích thước khung nhìn (viewport) của màn hình. Bài viết này giúp bạn tìm hiểu một số phương pháp để tạo hiệu ứng, cuộn ảnh nền và bản đồ trong game.

1. Ảnh nền nhiều tầng

Một phương pháp đơn giản và tạo hiệu ứng đẹp là sử dụng ảnh nền nhiều tầng. Bằng cách cho mỗi tầng có tốc độ cuộn khác nhau, khung cảnh trong game trở nên có chiều sâu và thật hơn. Số lượng tầng này thường chỉ giới hạn từ 2 đến 3 và các tầng nằm bên dưới sẽ có tốc độ cuộn chậm hơn.



2. Cuộn giả

Bạn có thể thấy nhiều game có bản đồ rất lớn nhưng ảnh nền chỉ lặp đi lặp lại theo một mẫu duy nhất. Điều này được thực hiện bằng cách sử dụng một ảnh nền có kích thước nhỏ và được vẽ lặp đi lặp lại trên viewport. Tùy theo kích thước của ảnh nền này mà ta sử dụng phương pháp vẽ khác nhau.

Cách thông thường là sử dụng một ảnh nền có kích thước bằng viewport. Sau đó ta chia ảnh nền này thành hai phần theo chiều dọc dựa vào một đường phân cách:
- Cắt phần ảnh nền từ vị trí đường phân cách đến hết và vẽ lên viewport tại vị trí {0,0}.
- Cắt phần ảnh nền từ vị trí đầu tiên đến đường phân cách (phần còn lại) và vẽ lên viewport tại vị trí tiếp nối với phần lúc nãy.

```
// position that divide the background into two parts
offsetX++;
if(offsetX>width)
        offsetX = 0;

// draw the first part
ctx.drawImage(bgimg,offsetX,0,width-offsetX,height,0,0,width-offsetX,height);
// the second part
ctx.drawImage(bgimg,0,0,offsetX,height,width-offsetX,0,offsetX,height);
```

3. … và cuộn thật

Với một bản đồ lớn, việc cuộn để giữ nhân vật chính của game luôn ở giữa màn hình rất đơn giản. Ta có thể xác định được tọa độ (left và top) trên bản đồ sẽ được dùng để làm viewport bằng cách:

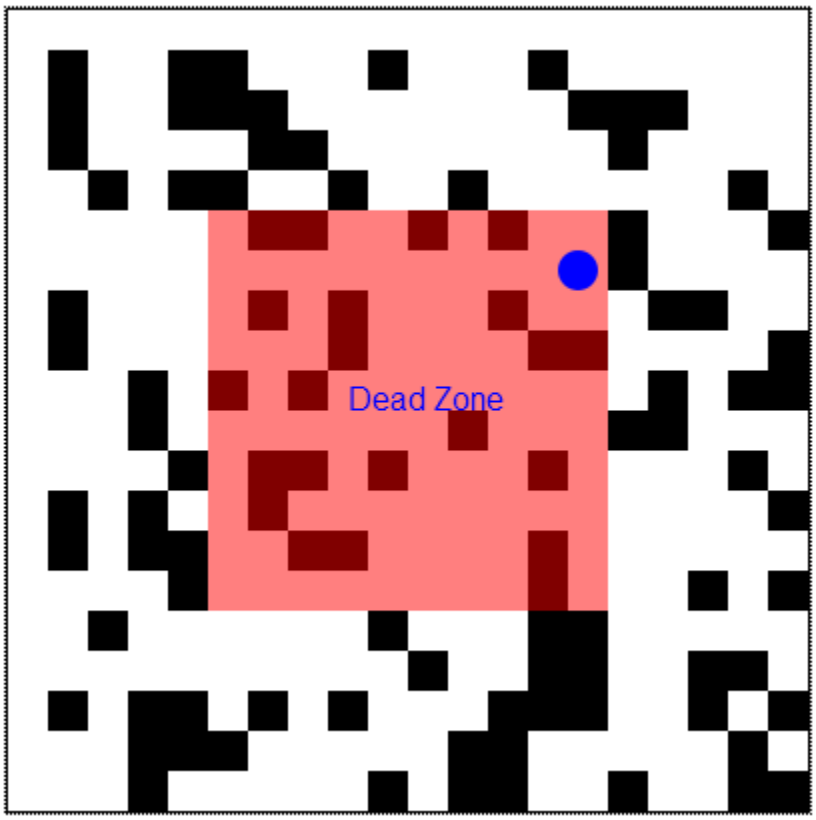
```
// inside Map object
// obj = character
this.offsetX = obj.left - viewWidth/2;
this.offsetY = obj.top - viewHeight/2;
```

Tuy nhiên việc thay đổi khung nhìn liên tục có thể khiến cho người chơi nhức mắt, mất tập trung và ảnh hưởng đến hiệu suất. Vì vậy ta áp dụng một giải phải là tạo một vùng giới hạn trong viewport gọi là “dead zone“. Khi nhân vật di chuyển nhưng vẫn nằm trong vùng này, viewport sẽ không bị thay đổi.

```
// inside Map object

var dx = obj.left - _map.offsetX;
var dy = obj.top - _map.offsetY;

if(dx<this.deadzone.left)
```

```
        this.offsetX = obj.left - this.deadzone.left;

else if(dx+obj.size>this.deadzone.right)

        this.offsetX = obj.right - this.deadzone.right;

if(dy<this.deadzone.top)
        this.offsetY = obj.top -  this.deadzone.top;
else if(dy+obj.size>this.deadzone.bottom)
        this.offsetY = obj.bottom - this.deadzone.bottom;
```

Xem Demo



II. Tạo Amimated Sprite

Sprite là một phương pháp để tạo các đối tượng chuyển động từ một hình ảnh duy nhất. Bằng cách sắp xếp nhiều đối tượng theo thứ tự chuyển động, sprite giúp cho việc quản lý tài nguyên và xử lý hiệu quả hơn so với việc phải sử dụng nhiều tập tin ảnh riêng lẻ.

Xem Demo.

Một ảnh có thể gồm nhiều sprite có cùng kích thước hợp lại với nhau. Ví dụ hình dưới đây ta có thể thấy mỗi hàng là một sprite hiển thị chuyển động của nhân vật theo phương khác nhau. Vậy

với mục đích là tạo lớp Sprite linh động, ta phải cho phép nó kết hợp được nhiều sprite với nhau và chuyển đổi qua lại giữa các sprite dễ dàng.

Mã nguồn của lớp AnimatedSprite:

```
var AnimatedSprite = function(data) {
	this.init(data);
	this.isFinished = false;
	this.currentSprite = null;
	this.currentFrame = 0;
	this.lastTick = 0;
};

AnimatedSprite.prototype = {
	start: function(){
		this.isFinished = false;
		this.currentFrame = 0;
	}
	init: function(data)
	{
		if(data){

			this.isLooping = data.isLooping;
			if(typeof this.isLooping!="boolean")
				this.isLooping = true;

			this.image = data.image;
			this.frameWidth = data.frameWidth;
			this.frameHeight = data.frameHeight || this.frameWidth;

			this.sprites = [];
			this.interval = data.interval;

			this.left = data.left;
			this.top = data.top;
			this.width = data.width || this.frameWidth;
			this.height = data.hegiht || this.frameHeight;

			this.onCompleted = data.onCompleted;
		}
	},
	addSprite: function(data){
		this.sprites[data.name]={
```

```
                        name : data.name,
                        startFrame : data.startFrame || 0,
                        framesCount : data.framesCount || 1,
                        framesPerRow                                    :
Math.floor(this.image.width/this.frameWidth)
                };

                this.currentSprite      =       this.currentSprite      ||
this.sprites[data.name];
        },
        setSprite: function(name){
                if(this.currentSprite != this.sprites[name])
                {
                        this.currentSprite = this.sprites[name];
                        this.currentFrame = 0;
                }
        },
        update: function(){
                if(this.isFinished)
                        return;
                var newTick = (new Date()).getTime();
                if(newTick-this.lastTick>=this.interval)
                {
                        this.currentFrame++;

                        if(this.currentFrame==this.currentSprite.framesCount)
                        {
                                if(this.isLooping)
                                        this.currentFrame=0;
                                else
                                {
                                        this.isFinished = true;
                                        if(this.onCompleted)
                                                this.onCompleted();
                                }

                        }
                        this.lastTick = newTick;
                }

        },
        draw: function(context){

                if(this.isFinished)
                        return;
                var realIndex =
this.currentSprite.startFrame+this.currentFrame;
                var row =
Math.floor(realIndex/this.currentSprite.framesPerRow);
                var col = realIndex % this.currentSprite.framesPerRow;


        context.drawImage(this.image,col*this.frameWidth,row*this.frameHeight,

        this.frameWidth,this.frameHeight,this.left,this.top,this.width,this.he
ight);
```

```
        }
}
```

Để sử dụng, ta có thể tạo một lớp con của AnimatedSprite để cung cấp các chức năng di chuyển, kiểm tra va chạm. Ở đây ta tạo một lớp Character thừa kế từ AnimatedSprite:

```
function Character(data){

        this.mapWidth = data.mapWidth;
        this.mapHeight = data.mapHeight;

        this.speedX = 0;
        this.speedY = 0;

        this.init(data);
}

Character.prototype = new AnimatedSprite();
Character.prototype.update = function(){

        var left = this.left+this.speedX;
        var top = this.top+this.speedY;
        var right = left+this.frameWidth;
        var bottom = top+this.frameHeight;

        if(left>0 && right<this.mapWidth)
                this.left = left;
        if(top>0 && bottom<this.mapHeight)
                this.top = top;
        AnimatedSprite.prototype.update.call(this);
};
```

Sau đó tạo đối tượng từ lớp Character ảnh bên trên để minh họa:

```
sprite = new Character({
        mapWidth: canvas.width,
        mapHeight: canvas.height,
        image: image,
        frameWidth: 32,
        frameHeight: 48,
        interval: 100,
        left: 10,
        top: 10,
});

sprite.addSprite({
        name: "walk_down",
        startFrame: 0,
        framesCount: 4
});

sprite.addSprite({
        name: "walk_left",
        startFrame: 4,
        framesCount: 4
});
```

```
sprite.addSprite({
        name: "walk_right",
        startFrame: 8,
        framesCount: 4
});
sprite.addSprite({
        name: "walk_up",
        startFrame: 12,
        framesCount: 4
});
sprite.setSprite("walk_left");
setInterval(function(){
        sprite.update();
        canvas.width = canvas.width;
        sprite.draw(context);
},1000/FPS);
```

III. Nạp trước hình ảnh và tài nguyên

Đối với những game sử dụng nhiều hình ảnh (âm thanh, video,…), ta cần phải nạp toàn bộ các ảnh cần thiết trước khi vào game. Bài viết này cung cấp một giải pháp đơn giản cho việc nạp trước các hình ảnh để sử dụng trong một canvas game.

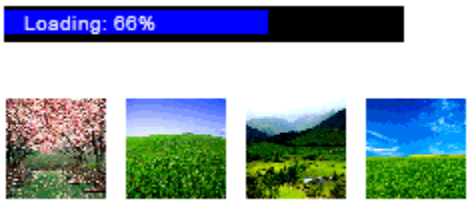
Từ ví dụ ở trang HTML5 Canvas Image Loader Tutorial, ta bổ sung thêm event cho phép cập nhật tiến trình nạp ảnh (onProgressChanged). Dựa vào cách này, bạn cũng có thể thay đổi để giúp việc nạp và quản lý các loại tài nguyên khác được thuận tiện:

```
function ImgLoader(sources,onProgressChanged,onCompleted) {
        this.images = {};
        var loadedImages = 0;
        var totalImages = 0;
        // get num of sources
        if(onProgressChanged || onCompleted)
                for(var src in sources) {
                        totalImages++;
                }
         var self = this;
        for(var src in sources) {
          this.images[src] = new Image();
          this.images[src].onload = function() {
                loadedImages++;
                if(onProgressChanged)
                {
                        var                         percent                         =
Math.floor((loadedImages/totalImages)*100);
                        onProgressChanged(this,percent);
                }
                if(onCompleted && loadedImages >= totalImages)
                        onCompleted(self.images);
          }
          this.images[src].src = sources[src];
        }
}
```

Cách sử dụng:

```
window.onload = function(images) {
        var canvas = document.getElementById("myCanvas");
        var context = canvas.getContext("2d");
        var barWidth = 200;
        var barLeft = (canvas.width-barWidth)/2;
        context.fillRect(barLeft,10,barWidth,18);
        context.fillStyle = "blue";
        var sources = {
          img1: "Spring/img (1).jpg",
          img2: "Spring/img (2).jpg",
          img3: "Spring/img (3).jpg",
          img4: "Spring/img (4).jpg",
          img5: "Spring/img (5).jpg",
          img6: "Spring/img (6).jpg",
        };
        var left = 100;

        var foo = new ImgLoader(sources,
                function(image,percent) {
                        context.drawImage(image, left, 55, 50,50);
                        //context.clearRect(0,0,200,30);
                        context.fillStyle = "blue";
                        context.fillRect(barLeft,12,percent*barWidth/100,12);
                        context.fillStyle = "white";
                        context.fillText("Loading:
"+percent+"%",barLeft+10,22);
                        left+=60;
                },
                function(images){
                        // completed
                }
        );
};
```

Minh họa:

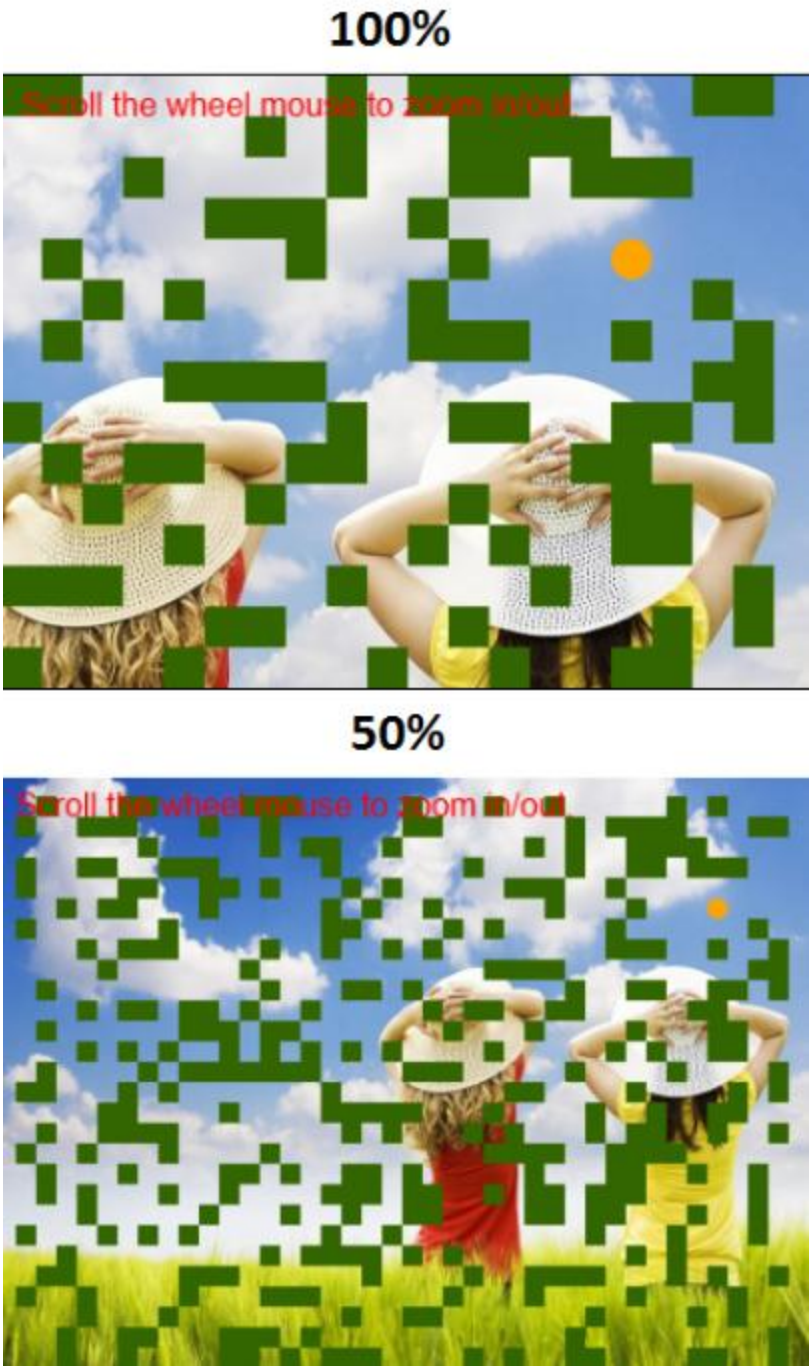


IV. Phóng to/thu nhỏ game bằng nút cuộn chuột

Để thay đổi kích thước nội dung trong canvas, cách tốt nhất là sử dụng phương thức context.scale(double x, double y) để thiết lập một tỉ lệ co giãn khi vẽ bằng context. Phương thức này giúp hình ảnh trong canvas vẫn giữ được độ nét như với các ảnh vector (không như việc bạn thay đổi kích thước của canvas bằng CSS hoặc chức năng zoom của trình duyệt). Bởi vì cách trên quá đơn giản nên ta sẽ dừng lại tại đây và tập trung vào phần xử lý thủ công. Qua đó, bạn có thể áp dụng cho những nền tảng game khác không hỗ trợ tranformation.

Bạn có thể áp dụng cách làm trong ví dụ này cho những ứng dụng tương tự, chủ yếu là thay đổi kích thước của một đơn vị (CELL_SIZE), của đối tượng và thay đổi tọa độ.

Xem Demo



1. Sự kiện Mouse Wheel trong javascript

Hoạt động của nút cuộn chuột được xác định bởi một giá trị delta lấy được từ tham số event của sự kiện mousewheel (hoặc DOMMouseScroll trên Firefox). Giá trị này cho biết chiều và độ lớn vòng quay của bánh xe. Bạn có thể đọc một bài hướng dẫn cụ thể tại Mouse wheel programming in JavaScript.

Trong ví dụ này ta chỉ quan tâm đến chiều quay của bánh xe: với giá trị âm, ta giảm kích thước nội dung bên trong canvas đi 1/2 và với giá trị dương ta tăng nó lên 2 lần.

```
function canvas_mousewheel(e){
        var delta = 0;
        if (!e) /* For IE. */
                e = window.e;
        if (e.wheelDelta) { /* IE/Opera. */
                        delta = e.wheelDelta/120;
        } else if (e.detail) { /** Mozilla case. */
                delta = -e.detail/3;
        }
        if (delta)
        {
                _map.changeZoom(delta);
                _ball.setZoom(_map.zoom);
                draw();
        }

        if (e.preventDefault)
                        e.preventDefault();
}
```

2. Thay đổi kích thước bản đồ

Trong các ví dụ sử dụng bản đồ, ta định nghĩa một hằng CELL_SIZE quy định kích thước của một ô theo pixel. Như vậy để thay đổi kích thước bản đồ không có gì phức tạp.

Trong ví dụ này ta tạo một thuộc tính zoom=1 xác định tỉ lệ kích thước ban đầu. Giá trị tối thiểu nó có thể nhận được là 0.5 và lớn nhất là 4:

```
Map.prototype = {
        changeZoom : function(zoom){

                if(zoom == 0 || (zoom<0 && this.zoom<=0.5) || (zoom>0 &&
this.zoom>=4))
                        return;

                this.zoom *= zoom < 0 ? 0.5 : 2;

                this.reset();
        },
        reset : function()
        {
                this.cellSize = CELL_SIZE*this.zoom;
```

```
                this.width = MAP_WIDTH*this.cellSize ;
                this.height = MAP_HEIGHT*this.cellSize ;

        }
        // ...
}
```

Phương thức reset() trên thực hiện cập nhật ba thuộc tính quan trọng của bản đồ khi thay đổi tỉ lệ kích thước.

3. Vẽ từng vùng bản đồ

Với dữ liệu của bản đồ được lưu trữ trong một mảng hai chiều. Ta cần xác định được vị trí, số lượng dòng và cột cần được hiển thị và chỉ vẽ vùng này lên canvas.

```
// draw method
var col = Math.floor(this.offsetX/this.cellSize);
var row = Math.floor(this.offsetY/this.cellSize);
ctx.fillStyle = "black";

for(var i=col;i<col+Math.floor(this.viewWidth/this.cellSize)+1;i++)
{
        if(this.data[i])
        for(var j=row;j<row+Math.floor(this.viewHeight/this.cellSize)+1;j++)
        {

                if(this.data[i][j]==1)
                        ctx.fillRect(i*this.cellSize                            -
this.offsetX,j*this.cellSize -this.offsetY,this.cellSize ,this.cellSize );
        }
}
```

Nếu sử dụng ảnh nền để vẽ, bạn cần chia tọa độ và độ lớn của vùng ảnh cho tỉ lệ zoom. Với tỉ lệ zoom càng lớn, vùng ảnh được vẽ càng nhỏ và ngược lại:

```
ctx.drawImage(this.background,this.offsetX/this.zoom,this.offsetY/this.zoom,t
his.viewWidth/this.zoom,this.viewHeight/this.zoom,0,0,this.viewWidth,this.vie
wHeight);
```

4. Áp dụng cho các nhân vật trên bản đồ

Mỗi nhân vật hay đối tượng được thêm vào bản đồ cũng cần được thay đổi kích thước và vị trí tương ứng với tỉ lệ zoom. Ta có thể tính được tọa độ (left, top) mới của đối tượng bằng cách:

new_left = old_left/old_zoom*new_zoom
new_top = old_top/old_zoom*new_zoom

```
this.setZoom = function(zoom){
        if(this.zoom!=zoom)
        {
```

```
                this.size = BALL_SIZE*zoom;
                this.left = this.left/this.zoom*zoom;
                this.top = this.top/this.zoom*zoom;
                this.right = this.left + this.size;
                this.bottom = this.top + this.size;
                this.zoom = zoom;
        }
};
```

V. Thay đổi kích thước Canvas theo trình duyệt

Một trong những yêu cầu thường thấy của game là thay đổi kích thước cửa cửa sổ game theo trình duyệt. Việc thay đổi này cần đảm bảo tốc độ của game không bị ảnh hưởng cũng như hình ảnh không bị biến dạng (giữ nguyên tỉ lệ chiều cao vào rộng).

Một trong những lợi thế của các trình duyệt hiện đại là hỗ trợ tính năng hardware accelerated cho canvas. Với cùng một pixel, trình duyệt có thể hiển thị canvas ở nhiều kích thước khác nhau.

Ví dụ như bạn gán canvas.width = 200 và canvas.height = 100, như vậy canvas có tất cả 20000 pixel. Và bạn có thể tăng gấp đôi kích thước của canvas lên nhưng lượng pixel mà bạn thao tác với canvas vẫn chỉ là 20000. Hay nói cách khác, việc xử lý và vẽ canvas với các kích thước khác nhau đã được bộ xử lý đồ họa (GPU) thực thi một cách tự động.

Ta phân biệt **kích thước thực** và **kích thước hiển thị** của canvas. Kích thước thực của canvas được gán thông qua hai thuộc tính là width và height:

```
canvas.width = 200;
canvas.height = 100;
```

Kích thước hiển thị của canvas được xác định thuộc tính style.width và style.height (bằng javascript hoặc CSS):

```
canvas.style.width = “400px”;
canvas.style.height = “200px”;
```

Độ nét của canvas phụ thuộc vào kích thước thực của nó, vì vậy hãy giữ một tỉ lệ vừa phải giữa hai loại kích thước này.

1. Điều chỉnh canvas thao kích thước trình **duyệt**

Một ví dụ đơn giản để bạn áp dụng khi cần thiết. Không chỉ cho canvas, bạn có thể sử dụng cách này để hiển thị hình ảnh, video,… (như xem ảnh slideshow).


```
function fitSize(canvas){
      var ratio = canvas.width/canvas.height;
```

```
	var width = window.innerWidth-5;
	var height = window.innerHeight-5;

	if(width/height>ratio)
		width = height*ratio;
	else
		height = width/ratio;

	canvas.style.width = width;
	canvas.style.height = height;

	canvas.style.top = (window.innerHeight-height)/2;
	canvas.style.left = (window.innerWidth-width)/2;
}
```

VI. Sử dụng Full Screen API

1. Giới thiệu

HTML5 cho phép bạn có thể đưa một thành phần/thẻ của trang vào trạng thái hiển thị fullscreen (khác với chế độ fullscreen của trình duyệt (F11)). Ngoài ra, bạn có thể sử dụng CSS để thay đổi cách hiển thị của thành phần khi nó ở trong trạng thái fullscreen.

API này gói gọn trong 5 mục sau:

- **element.requestFullScreen():** Phương thức kích hoạt trạng thái fullscreen của một thành phần trong trang web.
- **document.cancelFullScreen():** Phương thức hủy bỏ trạng thái fullscreen.
- **document.fullscreenchange**: Sự kiện được kích hoạt khi trạng thái fullscreen thay đổi.
- **document.fullScreen:** Thuộc tính boolean dùng để kiểm tra trạng thái fullscreen.
- **:full-screen**: Một pseudo-class của CSS dùng để định nghĩa cách hiển thị của thành phần.

Hiện tại API này chỉ mới được hỗ trợ trên Mozilla (Firefox) và WebKit (Chrome/Safari). Bạn cần sử dụng các tiền tố (vendor) là webkit và moz cho mỗi phương thức hay thuộc tính, style bên trên để áp dụng cho từng loại trình duyệt.
Nếu cần thiết, bạn có thể thêm trước hai vendor ms và o để chúng có thể hoạt động được sau khi IE, Opera tích hợp API này:

Phương thức enterFullScreen():

```
function enterFullScreen(id){

	var element = document.getElementById(id);

	element.requestFullscreen ? element.requestFullscreen() :
		element.mozRequestFullScreen ? element.mozRequestFullScreen()
:
		element.webkitRequestFullScreen ?
element.webkitRequestFullScreen() :
		alert("Fullscreen API is not supported in this browser");
```

```
}
```

Phương thức exitFullScreen():

```
function exitFullScreen(){

	document.exitFullscreen ? document.exitFullscreen() :
		document.mozCancelFullScreen  ?  document.mozCancelFullScreen()
:
		document.webkitCancelFullScreen                                  ?
document.webkitCancelFullScreen() :
		alert("Fullscreen API is not supported in this browser");

}
```

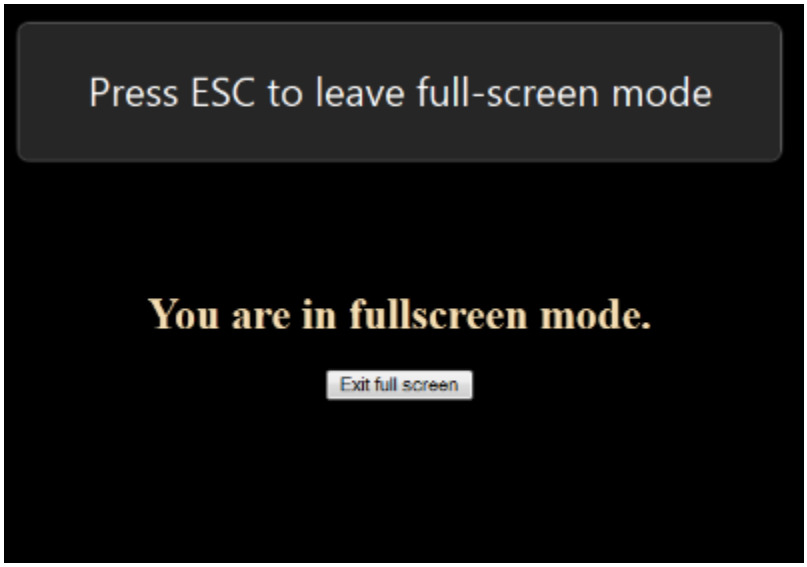
Sự kiện fullscreenchange và thuộc tính fullScreen:

```
document.addEventListener("fullscreenchange", function (e) {
	console.log(document.fullscreen);
});
document.addEventListener("mozfullscreenchange", function (e) {
	console.log(document.mozFullScreen);
});
document.addEventListener("webkitfullscreenchange", function (e) {
	console.log(document.webkitIsFullScreen);
});
```

CSS:

```
div:full-screen {
    background-color: gray;
}
div:-moz-full-screen {
    background-color: gray;
}
div:-webkit-full-screen {
    background-color: gray;
}
```

2. Ví dụ




```
<HTML>
<head>
<style>
#mydiv {
        padding-top: 200px;
        display: none;
        height: 400px;
        background: black;
}

#mydiv:full-screen { display: block; }
:full-screen h1 {  color: wheat; }

#mydiv:-moz-full-screen {  display: block; }
:-moz-full-screen h1{ color: wheat; }

#mydiv:-webkit-full-screen {  display: block; }
:-webkit-full-screen h1{ color: wheat; }

</style>
<script type="text/javascript">
function enterFullScreen(id){

        var element = document.getElementById(id);

        element.requestFullscreen ? element.requestFullscreen() :
                element.mozRequestFullScreen  ?  element.mozRequestFullScreen()
:
                element.webkitRequestFullScreen                                 ?
element.webkitRequestFullScreen() :
                alert("Fullscreen API is not supported in this browser");

}
function exitFullScreen(){
```

```
        document.exitFullscreen ? document.exitFullscreen() :
                document.mozCancelFullScreen  ?  document.mozCancelFullScreen()
:
                document.webkitCancelFullScreen ?
document.webkitCancelFullScreen() :
                alert("Fullscreen API is not supported in this browser");

}
document.addEventListener("fullscreenchange", function (e) {
        console.log(document.fullscreen);
});

document.addEventListener("mozfullscreenchange", function (e) {
        console.log(document.mozFullScreen);
});

document.addEventListener("webkitfullscreenchange", function (e) {
        console.log(document.webkitIsFullScreen);
});

</script>
</head>
<body>
<center>
        <div id="mydiv">
                <h1>You are in fullscreen mode.</h1>
                <input type="button" value="Exit full screen"
onclick="exitFullScreen('mydiv')"     />
        </div>
        <input type="button" value="Enter full screen"
onclick="enterFullScreen('mydiv')"/>
 </center>

</body>
</HTML>
```

VII. Tạo menu và chuyển đổi giữa các màn hình Game

Mỗi game được làm ra đều cần có màn hình chào mừng và các menu để giúp người chơi chuyển đổi giữa các màn hình khác nhau như: giới thiệu, chơi game, hướng dẫn, tùy chọn. Từ yêu cầu của một bạn, ta sẽ hướng dẫn cách thực hiện phần này một cách đơn giản và nhanh chóng.

Xem Demo

1. Lớp MenuItem

Mỗi đối tượng MenuItem tương ứng với một mục menu trên màn hình. Một đối tượng thế này có thể chỉ cần 4 giá trị xác định tọa độ và kích thước. Tuy nhiên để dễ thay đổi và phát triển hơn. Ta tạo thêm một số thuộc tính và cung cấp một sự kiện onclick cho nó.

Dựa vào thuộc tính isMouseOver, ta sẽ thay đổi màu sắc hiển thị của MenuItem mỗi khi chuột được hover. Phần này khá đơn giản, chỉ cần coi đây là một cấu trúc dữ liệu để lưu các giá trị của một hình chữ nhật cùng với phương thức vẽ để hiển thị nó.

```
function MenuItem(data){
	this.left = data.left || 0;
	this.top = data.top || 0;
	this.width = data.width || 100;
	this.height = data.height || 30;
	this.text = data.text || "Menu Item";
	this.onclick = data.onclick;

	this.right = this.left+this.width;
	this.bottom = this.top+this.height;
	this.centerX = this.left+this.width/2;
	this.centerY = this.top+this.height/2;

	this.isMouseOver = false;

	this.update = function(){

	};
	this.draw = function(context){
		context.font = "16px Arial";
		context.textAlign = "center";
		if(this.isMouseOver)
			context.fillStyle = "rgba(255,255,255,0.7)";
		else
			context.fillStyle = "rgba(255,255,255,0.2)";

		context.fillRect(this.left,this.top,this.width,this.height);

		context.fillStyle = "white";
		context.fillText(this.text,this.centerX,this.centerY);
		context.strokeRect(this.left,this.top,this.width,this.height);
	};
```

```
	this.contain = function(x,y){
		return !(x<this.left || x>this.right || y<this.top || y>this.bottom);
	};
}
function MenuItem(data){
	this.left = data.left || 0;
	this.top = data.top || 0;
	this.width = data.width || 100;
	this.height = data.height || 30;
	this.text = data.text || "Menu Item";
	this.onclick = data.onclick;

	this.right = this.left+this.width;
	this.bottom = this.top+this.height;
	this.centerX = this.left+this.width/2;
	this.centerY = this.top+this.height/2;

	this.isMouseOver = false;

	this.update = function(){

	};
	this.draw = function(context){
		context.font = "16px Arial";
		context.textAlign = "center";
		if(this.isMouseOver)
			context.fillStyle = "rgba(255,255,255,0.7)";
		else
			context.fillStyle = "rgba(255,255,255,0.2)";

		context.fillRect(this.left,this.top,this.width,this.height);

		context.fillStyle = "white";
		context.fillText(this.text,this.centerX,this.centerY);
		context.strokeRect(this.left,this.top,this.width,this.height);
	};
	this.contain = function(x,y){
		return !(x<this.left || x>this.right || y<this.top || y>this.bottom);
	};
}
```

2. Lớp Screen

Lớp này là phần chính của tạo nên một game. Một game có thể được tạo ra từ một hoặc nhiều Screen và người chơi có thể chuyển qua lại giữa các Screen bằng một vài nút bấm nào đó (trong ví dụ này là MenuItem).

Lớp Screen này là nơi quản lý chính và có thể coi là một game thu nhỏ trong ứng dụng. Giống như mọi ví dụ và mọi đối tượng game ta từng thực hiện, lớp này sẽ gồm hai phương thức chính là update() và draw(). Bên cạnh đó ta có thể bổ sung các phương thức để bắt đầu và kết thúc

game. Ví dụ như start() và stop(). Vì đây là ví dụ tập trung vào việc thiết kế menu và chuyển đổi giữa các màn hình, các đối tượng Screen ta tạo chỉ chứa các MenuItem.

Chú ý: Vì mỗi lớp Screen sẽ đăng kí các event onmousemove, onclick cho canvas, nên khi start() một Screen, bạn cần phải đăng kí các event này lại các event này. Nếu không nó sẽ bị overwrite bởi event từ các Screen khác.

```
var CELL_SIZE = 10;
var FPS = 10   ;
var WIDTH = 400;
var HEIGHT = 400;

function Screen(canvas){
        var timer;
        var width = canvas.width;
        var height = canvas.height;
        var context = canvas.getContext("2d");
        this.items = [];
        // this method/event is actived in the end of draw() method
        this.afterDraw = null;

        this.update = function(){
                for(var i=0;i<this.items.length;i++){
                        this.items[i].update();
                }
        };
        this.draw = function(){

                context.fillStyle = "black";
                context.fillRect(0,0,width,height);
                for(var i=0;i<this.items.length;i++){
                        this.items[i].draw(context);
                }

                if(this.afterDraw)
                        this.afterDraw(context);
        };
        this.start = function(){
                this.stop();
                var self = this;
                // register events
                canvas.onclick = function(e){
                        // raise the onclick event of each MenuItem when it is
clicked
                        var x = e.pageX - this.offsetLeft;
                        var y = e.pageY - this.offsetTop;
                        for(var i=0;i<self.items.length;i++){
                                if(self.items[i].onclick                        &&
self.items[i].contain(x,y))
                                        self.items[i].onclick(x,y);
                        }
                };
                canvas.onmousemove = function(e){
                        // change the isMouseOver property of each MenuItem
                        var x = e.pageX - this.offsetLeft;
```

```
                        var y = e.pageY - this.offsetTop;
                        canvas.style.cursor = 'default';
                        for(var i=0;i<self.items.length;i++){
                                self.items[i].isMouseOver                       =
self.items[i].contain(x,y);
                                // change the cursor type to hand
                                if(self.items[i].isMouseOver)
                                        canvas.style.cursor = 'pointer';
                        }
                };
                timer = setInterval(function(){
                        self.update();
                        self.draw();
                },1000/FPS);
        };
        this.stop = function(){
                if(timer)
                        clearInterval(timer);
                timer = null;
        };
        this.addItem = function(item){
                this.items.push(item);
        };
}
```

3. Kiểm tra kết quả

Vậy là xong hai lớp chính của ví dụ, bạn tạo một file HTML với tên bất kì để test với nội dung bên dưới.

F. AI trong game

I. Giới thiệu

AI hay trí tuệ nhân tạo được sử dụng trong rất nhiều game, từ đơn giản đến phức tạp. Tùy theo từng trường hợp mà AI có thể tính toán vị trí để di chuyển, tìm đường đi, phân tích trạng thái để đưa ra quyết định,…

Một trong những game thuộc loại AI đơn giản nhất là trò chơi Pong được phát hành năm 1970. Dạng game bóng bàn này cho phép người chơi đấu với máy bằng cách điều khiển một thanh paddle dọc để đón và đánh trả bóng. Và bạn cũng có thể hình dung được AI của máy hoạt động như thế nào: chỉ cần thay đổi vị trí của paddle lên xuống theo trái bóng.

Sau này, những thể loại game mới ra đời đòi hỏi trình độ AI phải nâng lên một tầm mới. Thường thấy nhất là các thể loại game với thuật toán tìm kiếm trên cấu trúc dữ liệu dạng cây để tìm đường đi hay đưa ra quyết định (decision tree). Trong các thuật toán duyệt cây, tùy theo trường hợp và độ lớn của dữ liệu, lập trình viên có thể thay đổi và chọn lựa thuật toán như Hill climbing, Breadth First Search, Depth First Search, Best First Search, Iterative Depth First Search hay thậm chí có thể kết hợp một, hai loại thuật toán với nhau.

Với game Rắn Săn Mồi, một ý tưởng của ta là áp dụng AI trong game này để người chơi có thể đấu với máy. Như vậy bài toán đặt ra là phải làm sao để con rắn mà máy điều khiển có thể tìm được đường ngắn nhất đến thức ăn mà không lao đầu vào tường hoặc cắn nhầm thân của nó.

II. Phân tích để lựa chọn thuật toán

1. Không gian dữ liệu nhỏ:.

Ở đây chính là kích thước của bản đồ, với loại game này thì chỉ giới hạn trong khoảng 50×50 ô là đủ.

2. Cần tìm ra đường đi ngắn nhất.

Với hai yếu tố này, bạn dễ dàng nhận ra lựa chọn giải thuật tìm kiếm theo chiều sâu (Breadth First Search) là hợp lý nhất. Ta cũng có thể loại trừ để chọn ra được thuật giải đúng:
- Hill climbing hay Best First Search: không khả thi do cần một hàm lượng giá hay heuristic. Với loại game mà vị trí hay trạng thái nhân vật không bị ảnh hưởng nhiều bởi các yếu tố môi trường thì điều này không cần thiết, thậm chí có thể đưa ra kết quả sai.
- Depth First Search hay Iterative Depth First Search: tìm kiếm theo chiều sâu theo kiểu may rủi có thể khiến nhân vật "đi lạc" quá đà và làm chậm thời gian tìm thấy con đường chính xác. Hơn nữa, kết quả tìm được có thể không phải là đường đi ngắn nhất.

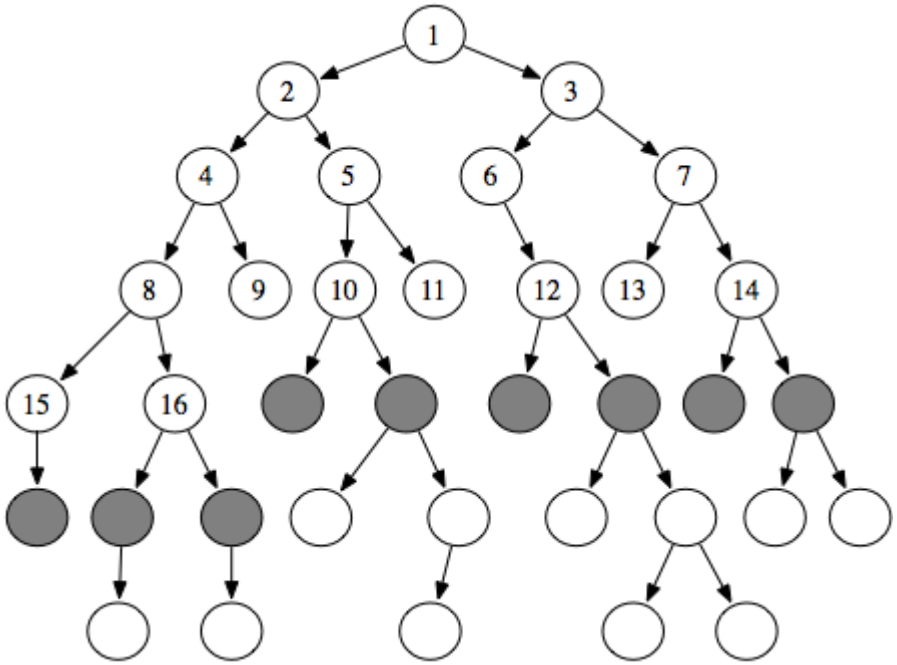
III. Thuật toán Breadth First Search

(Nếu bạn đã quen thuộc với thuật toán Breadth First Search, có thể bỏ qua phần này.) Cơ chế làm việc của thuật toán tương tự như vết dầu loang, tìm kiếm những điểm gần nhất trước. Bạn có thể thấy một vài game sử dụng bản đồ hay liên quan đến AI cũng có thể sử dụng thuật toán này. Một ví dụ bạn có thể áp dụng thuật toán này là n-puzzle mà ta đã cài đặt bằng thuật toán A* để giải quyết.

Trong giải thuật này ta cần định nghĩa các thành phần sau:

Open	Danh sách chứa các vị trí chờ được xét
Close	Danh sách chứa các vị trí đã xét.
start	Vị trí bắt đầu
goal	Vị trí kết thúc
n	vị trí đang xét.
G(n)	Tập các vị trí có thể đến từ vị trí n.

Hình minh họa (nguồn http://artint.info/html/ArtInt_54.html):



Giải thuật được mô tả bằng mã giả như sau:

```
Begin
        Open := {start};
        Close := ø;
        While (Open <> ø) do
                begin
                        n:= Dequeue(Open);
                        if (n=goal) then Return True;
                        Open := Open + G(n);
                        Close := Close + {n};
                end;
        Return False;
End;
```

Ta thấy rằng G(n) dựa vào tập Close để kiểm tra các vị trí có cần được kiểm tra hay không. Nếu bạn cài đặt hai tập này bằng một collection thì tốc độ thực thi sẽ tương đối chậm vì phải thực hiện tìm kiếm phần tử trong danh sách.

Do ví dụ của ta có dạng bản đồ nên thay vì dùng collection, ta sẽ sử dụng mảng hai chiều để có thể truy xuất trực tiếp đến một phần tử dựa vào vị trí. Khi đó ta có thể kiểm tra thuộc tính của phần tử xem nó đã được duyệt chưa.

IV. Các quy tắc trong game

Do có cách chơi khác với các game cùng loại nên ta cần đặt thêm một vài quy tắc để xây dựng game:
- Tốc độ di chuyển của (con rắn) người chơi và máy phải bằng nhau.
- Khi cả hai đến đồ ăn cùng lúc, ưu tiên cho người chơi lấy được nó.

- Hai con rắn có thể di chuyển xuyên qua nhau (nếu không chúng có thể cắn nhau và con bị cắn sẽ chết).
- Khi độ dài con rắn của máy đạt đến một mốc nào đó, người chơi sẽ thua cuộc.
- Để giảm mức độ khó, độ dài con rắn mà máy cần đạt được sẽ cao hơn người chơi.

Tạm ổn, trong phần tới ta sẽ thực hiện việc cài đặt game này trên HTML5-Canvas.

V. Xây dựng một lớp Queue dựa trên mảng

Việc sử dụng thuật toán Breadth First Search (BFS) cần phải sử dụng một cấu trúc dữ liệu kiểu hàng đợi (Queue) để làm tập Open.

Định nghĩa:

"Hàng đợi (tiếng Anh: queue) là một cấu trúc dữ liệu dùng để chứa các đối tượng làm việc theo cơ chế FIFO (viết tắt từ tiếng Anh: First In First Out), nghĩa là "vào trước ra trước Trong hàng đợi, các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào, nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi. Thao tác thêm vào và lấy một đối tượng ra khỏi hàng đợi được gọi lần lượt là 'enqueue' và 'dequeue'. Việc thêm một đối tượng luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi." (Wikipedia);

Trong javascript, ta có thể dễ dàng tạo một lớp Queue nhờ những phương thức sẵn có của mảng:

```
function Queue(){
	var data  = [];

	this.clear = function(){
		data.length = 0;
	}

	this.getLength = function(){
		return data.length;
	}

	this.isEmpty = function(){
		return data.length == 0;
	}

	this.enqueue = function(item){
		data.push(item);
	}

	this.dequeue = function(){
		if (data.length == 0) return undefined;
		return data.shift();
	}

	this.peek = function(){
		return (data.length > 0 ? data[0] : undefined);
	}
}
```

VI. Cài đặt thuật toán Breadth First Search

Tốc độ tìm đường đi trong game này rất quan trọng vì càng lên level cao, việc di chuyển của con rắn có thể lên tới 60 ô trong một giây (tương ứng FPS = 60) hay thậm chí có thể cao hơn. Nếu tốc độ quá chậm có thể khiến game bị "đứng" mỗi khi thuật toán này được sử dụng và tệ hơn là con rắn sẽ lao đầu vào tường như một chiếc xe mất phanh.

Nếu khi test bạn thấy tốc độ chưa được ưng ý, đặc biệt trên các máy yếu, hãy thử làm chậm tốc độ game lại và tăng thêm độ phức tạp của bản đồ. Một cách khác nữa là giảm kích thước bản đồ để giảm không gian tìm kiếm. Tuy nhiên bạn không cần quá lo lắng về tốc độ tìm đường, theo thử nghiệm của ta thì với FPS = 1000 và độ dài rắn lên tới 100, game vẫn chạy ổn và con rắn không để mất một "viên kẹo" nào.

Để cài đặt thuật toán này, trước tiên ta tạo một lớp Node dùng để lưu giữ thông tin của một phần tử trong mảng hai chiều. Ta cần hai thuộc tính x,y để lưu lại chính vị trí dòng và cột của Node trong mảng. Như vậy ta có thể xác định được ngay vị trí của một đối tượng Node trong mảng mà không cần lặp qua mảng để tìm kiếm (do Node được lưu trong Queue).

```
function Node(x,y,value){
	this.x = x;
	this.y = y;
	this.value = value;
	this.visited = false;
}
```

Mặc dù dễ hiểu nhưng cách này sẽ khiến việc tìm được chưa đạt được tốc độ mà ta mong muốn do phải so sánh 2 thuộc tính value và visited để xác định một Node đã được "ghé thăm" chưa. Vì vậy, ta sẽ bỏ đi thuộc tính visited và tận dụng thuộc tính value. Ta có lớp Node mới:

```
var BLANK = 0;
var WALL = 1;
var SNAKE = 2;
var VISITED = 3;

function Node(x,y,value){
	this.x = x;
	this.y = y;
	this.value = value;
}
```

Vòng lặp chính của thuật toán:

```
// ...
// add the start node to queue
open.enqueue(start);
// the main loop

while(!open.isEmpty())
{
	node = open.dequeue();
	if(node)
	{
		if(node.x==goal.x && node.y==goal.y)
```

```
                {
                        return getSolution(node);
                }
                genMove(node);
        }
        else
                break;
}
// ...
```

Phương thức sinh các nước đi tiếp theo tại vị trí đầu rắn:

```
// generate next states by adding neighbour nodes
function genMove(node)
{
        if (node.x < cols - 1)
                addToOpen(node.x + 1, node.y, node);
        if (node.y < rows - 1)
                addToOpen(node.x, node.y + 1, node);
        if (node.x > 0)
                addToOpen(node.x - 1, node.y, node);
        if (node.y > 0)
                addToOpen(node.x, node.y - 1, node);
}
```

Phương thức thêm một nước đi vào tập Open, ta chỉ cần thêm các node rỗng (BLANK). Khi thêm vào, ta đặt lại trạng thái của node là VISITED để không thêm nó vào lần thứ 2:

```
function addToOpen(x,y, previous)
{
        var node = nodes[x][y];

        if (node.value==BLANK)
        {
                // mark this node as visited to avoid adding it multiple times
                node.value = VISITED;
                // store the previous node
                // so that we can backtrack to find the optimum path
                // (by using the getSolution() method)
                node.previous = previous;
                open.enqueue(node);
        }
}
```

Phương thức dùng để lấy đường đi sau khi tìm được vị trí của node đích. Ta chỉ cần backtracking các node dựa vào thuộc tính previous của các node:

```
function getSolution(p)
{
        var nodes = [];
        nodes.push(p);
        while (p.previous)
        {
                nodes.push(p.previous);
                p = p.previous;
        }
```

```
        return nodes;
}
```

Như vậy, khi đã có đường đi đến đích, ta chỉ cần cho rắn di chuyển từng ô dựa theo đường đi này cho đến hết độ dài.

VII. Di chuyển đối tượng theo đường đi

Trong lớp Snake, ta tạo một biến autoMoving để xác định quyền điều khiển đối tượng thuộc về người chơi hay tự động (máy tính). Mỗi khi một viên thức ăn mới tạo ra, ta sẽ cập nhật lại đường đi mới cho con rắn. Chỉ cần thêm một dòng lệnh vào trong phương thức createFood().

```
function createFood() {
        var x = Math.floor(Math.random()*_cols);
        var y;
        do {
                y = Math.floor(Math.random()*_rows);
        }     while(_walls[x][y]     ||     _comSnake.contain(x,     y)     ||
_playerSnake.contain(x, y));

        _food = {x: x, y: y};
        // find new path for the com player
        _comSnake.setPath(_bfs.findPath(_comSnake.data,_comSnake.getHead(),_fo
od));
}
```

Trong lớp Snake, ta cần tạo một phương thức mới giúp rắn di chuyển tự động. Bởi vì việc di chuyển của rắn dựa vào bốn hướng (do còn cho phép người chơi điều khiển), ta có thể xác định hướng di chuyển dựa vào vị trí của ô cũ và mới của đầu rắn:

```
// Snake
this.move = function(){
        if(this.stepIndex>0)
        {
                this.stepIndex--;
                var newPos = this.path[this.stepIndex];
                if(newPos.x<this.data[0].x)
                        this.direction = LEFT;
                else if(newPos.x>this.data[0].x)
                        this.direction = RIGHT;
                else if(newPos.y<this.data[0].y)
                        this.direction = UP;
                else if(newPos.y>this.data[0].y)
                        this.direction = DOWN;
        }
};
```

VIII. Vòng lặp chính của game

Phần quan trọng nhất giúp game vận hành, phần này khá đơn giản và được chú thích nên ta cũng không cần giải thích thêm.

```
function update() {
	if(!_running)
		return;

	_playerSnake.handleKey(_pressedKey);
	// player has priority to eat
	var ret =_playerSnake.update(_walls,_food);
	if(ret==1) // player eated the food
	{
		_scores += _level*2;
		createFood();
	}
	else if(ret==2) // player collided with something
	{
		if(_scores>=0)
		{
			_scores -= _level*2;
			if(_scores<0)
				_scores = 0;
		}
		endGame();
		return;
	}else{
		if(!_comSnake.path)

	_comSnake.setPath(_bfs.findPath(_comSnake.data,_comSnake.getHead(),_fo
od),_food);

		ret = _comSnake.update(_walls,_food);
		if(ret==1) // com player eated the food
			createFood();
	}
	draw();
	// Player's snake reached the maximum length
	// so the game will start the next level
	if(_playerSnake.data.length==MAX_PLAYER_LENGTH)
	{
		// go to next level
		_level++;
		_scores += _level*100;
		_running = false;
		_context.save();
		_context.fillStyle = "rgba(0,0,0,0.2)";
		_context.fillRect(0,0,WIDTH,HEIGHT);
		_context.restore();
		_context.fillStyle = "red";
		_context.textAlign = "center";
		_context.fillText("Press Enter to start the next
level",WIDTH/2,HEIGHT/2);
	}else if(_comSnake.data.length==MAX_COM_LENGTH)
	{
		endGame();
		return;
	}
}
```

G. Một nền tảng game 2D side-scrolling

Từ phần Map Scrolling, ta có thể thấy đây là một sườn game rất phổ biến cho các loại game sử dụng bản đồ (như Pac-Man, Battle City hay Mario,…). Trong bài này ta sẽ thay đổi một vài đặc điểm để tạo một demo dạng game phiêu lưu màn hình ngang.

I. Cơ bản

1. Tạo bản đồ

Bản đồ được dùng trong loại game này có thể được xem như một lưới với mỗi ô chứa một giá trị đại diện cho loại đối tượng (thường là vật cản như tường, đá, cây, …). Để đơn giản, ta tạo một bản đồ chỉ gồm duy nhất một loại vật cản và được phân bố ngẫu nhiên:

map.js:

```
for(var i=1;i<MAP_WIDTH-1;i+=2)
{
        this.data[i] = [];
        this.data[i+1] = [];
        for(var j=1;j<MAP_HEIGHT;j++)
        {
                this.data[i][j] = Math.floor(Math.random()*6);
                this.data[i+1][j] = this.data[i][j];
                // create image buffer
                if(this.data[i][j]==BRICK)
                {

        context.fillRect(i*CELL_SIZE,j*CELL_SIZE,CELL_SIZE*2,CELL_SIZE);
                        context.fill();
                }
        }
}
```

2. Kiểm tra va chạm

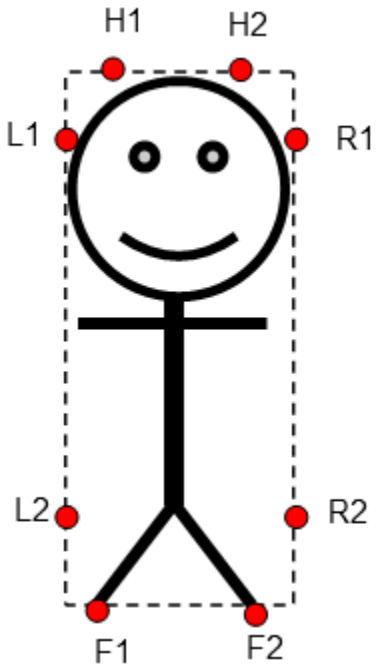
Ta có thể kiểm tra nhanh va chạm của nhân vật với các vật cản bằng cách xác định tọa độ ô tương ứng của bản đồ từ vị trí của nhân vật.

map.js:

```
this.contain = function(x,y){
        var col = Math.floor(x/CELL_SIZE);
        var row = Math.floor(y/CELL_SIZE);
        if(!this.data[col])
                return false;
        if(this.data[col][row]==BRICK)
        {
```

```
                var b = {
                        left: col*CELL_SIZE,
                        top: row*CELL_SIZE
                };
                b.right = b.left+CELL_SIZE;
                b.bottom = b.top+CELL_SIZE;
                return b;
        }
        return false;
}
```

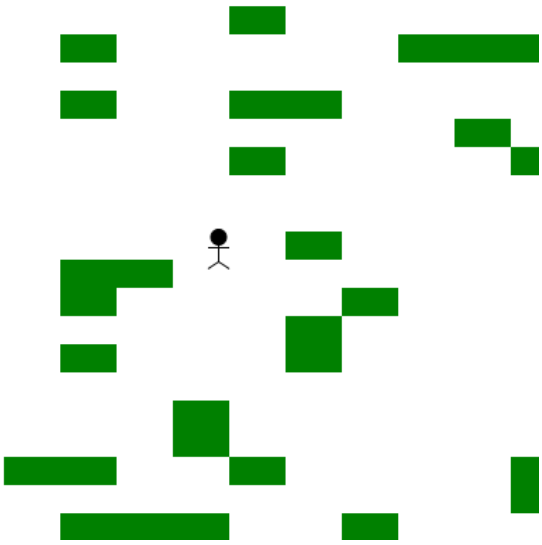
Đối với nhân vật, ta sử dụng 8 điểm bao quanh nhân vật để kiểm tra (các điểm màu đỏ trong hình sau). Muốn việc kiểm tra chính xác hơn, bạn chỉ cần tăng thêm số điểm tuy nhiên thời gian kiểm tra sẽ lâu hơn. Dựa vào các điểm này ta có thể xác định được va chạm và ngừng việc di chuyển của nhân vật theo các trường hợp các điểm va chạm là:



- R1 hoặc R2: Va chạm bên phải. Nếu speedX>0, ta gán speedX = 0.
- L1 hoặc L2: Va chạm bên trái. Nếu speedX – H1 hoặc H2: Nhân vật nhảy lên và đụng đầu vào vật cản. Ta dặt speedY = 0 và falling = true để nhân vật dừng lại và bắt đầu rơi.
- H1 hoặc H2: Nhân vật nhảy rớt xuống đất hoặc vật cản. Ta dặt speedY = 0 và falling = false.

```
Player.prototype.update = function(){
        if(this.isJumping)
                this.speedY += GRAVITY;

        var vtop = this.top+this.speedY;
        var vbottom = vtop+this.height;
        var vleft = this.left+this.speedX;
        var vright = vleft+this.width;

        if(this.isJumping)
                vbottom -= 1;
```

```
        var b;

        this.isJumping = true;

        if(vbottom >= this.map.height)
        {
                this.top = this.map.height-this.height;
                this.speedY = 0;

                this.isJumping = false;
        }
        else if(b = this.map.contain(this.left+2,vbottom) ||
this.map.contain(this.right-2,vbottom))
        {
                this.top = b.top-this.height;
                this.speedY = 0;
                this.isJumping = false;
        }
        else if(b = this.map.contain(this.left+2,vtop) ||
this.map.contain(this.right-2,vtop))
        {
                this.top = b.bottom;
                this.speedY = 0;
        }

        vtop = this.top+this.speedY;

        vbottom = vtop+this.height;

        if(this.left<=0 || (b = this.map.contain(vleft,vtop+6) ||
this.map.contain(vleft,vbottom-4)))
        {
                if(b)
                        this.left = b.right;

                if(this.speedX<0)
                        this.speedX = 0;
        }else if(this.right>=this.map.width || (b =
this.map.contain(vright,vtop+6) || this.map.contain(vright,vbottom-4)))
        {
                if(b)
                        this.left = b.left-this.width;

                if(this.speedX>0)
                        this.speedX = 0;
        }
        this.top = vtop;
        this.bottom = vbottom;
        this.left += this.speedX;
        this.right = this.left + this.width;
}
```



II. Thêm các chức năng và nhân vật

1. Lớp Character

Đây là lớp nền tảng cho tất cả cả đối tượng chuyển động (có thể thay đổi vị trí) trong game. Các đối tượng đó có thể là người chơi, quái vật hay các NPC (non-player character). Ta định nghĩa khá nhiều thuộc tính trong lớp này để xác định vị trí, tốc độ, kích thước của nhân vật. Bên cạnh đó, ta thêm một số thuộc tính boolean hành vi (hay khả năng) của nhân vật:

- canDestroyObstacles: khả năng phá hủy các chướng ngại vật (bằng cách dùng đầu) của nhân vật. Khi nhân vật nhảy lên, nếu đầu chạm trúng chướng ngại vật có thể phá hủy (gạch), ta sẽ kiểm tra thuộc tính này để xác định xem chướng ngại vật có bị phá hủy hay không.
- isAutoMoving: Dùng cho các nhân vật không bị người chơi (player) điều khiển (quái vật).
- canJump: Nhân vật có thể nhảy hay không.

```
function Character(map,options){

	if(!options)
		options = {};

	this.map = map;
	this.left = options.left || 0;
	this.top = options.top || 300;
	this.height = options.height || 15;
	this.width = options.width || 15;

	this.jumpPower = options.jumpPower || 5;
	this.speed = options.speed || 2;
	this.speedX = (Math.random()<0.5? -this.speed: this.speed);
```

```
        this.speedY = 0;

        this.isJumping = true;
        this.isDead = false;

        this.bottom = this.top+this.height;
        this.right = this.left+this.width;
        // behaviors
        this.canDestroyObstacles = options.canDestroyObstacles;
        this.isAutoMoving = options.isAutoMoving;
        this.canJump = options.canJump;

}
```

Nhìn chung phương thức chính update() cũng tương tự như trong bài part 0. Ngoại trừ việc khi nhân vật nhảy lên và phá gạch, các đối thủ (trong game này là các Monster) bên trên sẽ bị tiêu diệt. Vì vậy ta bổ sung đoạn mã này vào phần kiểm tra va chạm phía trên:

```
// top collision
if(this.canDestroyObstacles)
{
        // destroy all monsters that are standing on the brick
        for(m in this.map.monsters)
        {
                var mon = this.map.monsters[m];
                if(!mon)
                        continue;
                var cx = mon.left + mon.width/2;
                if(mon.bottom==b.top && cx>=b.left && cx<=b.right )
                        mon.die();
        }
        this.top = b.bottom;
        this.speedY = 0;
}
```

Thay đổi hướng di chuyển của nhân vật khi thuộc tính isAutoMoving có giá trị true và nhân vật bị va chạm ở phía trái hoặc phải:

```
// update() method
if(this.left<0 || (b = this.map.colllide(vleft,vtop+6,false) ||
        this.map.colllide(vleft,vbottom-4,false,this.canEat))) // left
{
        if(b && b!=true)
                this.left = b.right;

        if(this.speedX<0)
                this.speedX = this.isAutoMoving? this.speed: 0;

}else          if(this.right>=this.map.width          ||          (b          =
this.map.colllide(vright,vtop+6,false) ||
                this.map.colllide(vright,vbottom-4,false))) // right
{
        if(b && b!=true)
                this.left = b.left-this.width;

        if(this.speedX>0)
                this.speedX = this.isAutoMoving? -this.speed: 0;
}
```

2. Lớp Monster

Monster (hay Enemy) là các đối tượng có khả năng hoạt động và có thể được tích hợp AI nhằm tiêu diệt hoặc cản trở người chơi. Trong bất kì game nào thì các loại đối tượng này rất đa dạng và là thường là nguyên nhân chính tạo ra sự lôi cuốn của game. Vì đây chỉ là phần bắt đầu nên ta chỉ tạo một loại Monster duy nhất có khả năng tiêu diệt người chơi thông qua va chạm.

Bởi vì được thừa kế từ lớp Character bên trên, lớp này chỉ cần một công việc chính là hiện thực phương thức draw() để vẽ chính nó:

```
function Monster(map,left,top){
        // call the super-constructor
        Character.call(this,map,{
                left: left,
                top: top,
                width: 20,
                height: 20,
                speed: 1,
                isAutoMoving: true
        });
}

Monster.prototype = new Character();

Monster.prototype.draw = function(context){

        context.save();
        context.beginPath();

        var left = this.left-this.map.offsetX;
        var top = this.top-this.map.offsetY;

        var right = left+this.width;
        var bottom = top+this.height;

        var hw = this.width/2;
        var cx = left+hw;

        context.fillStyle = "violet";
        context.arc(cx,top+hw,hw-2,0,Math.PI*2,true);
        //context.rect(left,top,this.width,this.height);
        context.fill();

        context.stroke();
```

```
        context.restore();
}
```

3. Lớp Player

Cũng tương tự như Monster, tuy nhiên công việc sẽ phức tạp hơn vì ta cần kiểm tra nhiều thứ và phải cung cấp các phím bấm điều khiển. Phương thức update này mặc định trả về 0. Trường hợp player hoàn thành vòng chơi, trả về 1; và nếu player chết, trả về 2:

```
Player.prototype.update = function(){
        if(this.right>=this.map.width){
                alert("Game Over!");
                return 1;
        }
        if(this.isFalling) // die
        {
                this.speedY += GRAVITY;
                this.top += this.speedY;
                return (this.top>this.map.height)? 2 : 0 ;
        }

        Character.prototype.update.call(this);

        this.collide(this.map.monsters);
}
```

Kiểm tra va chạm với các monster, ta xem player và monster là các hình chữ nhật và chỉ cần kiểm tra xem hai hình này có cắt nhau không:

```
Player.prototype.collide = function(monsters){
        for(m in monsters)
        {
                var mon = monsters[m];
                if(!mon)
                        continue;
                if(!(this.left > mon.right ||
                        this.right < mon.left ||
                        this.top > mon.bottom ||
                        this.bottom < mon.top))
                        {
                                if(this.bottom<mon.bottom&& this.speedY>0)
                                {
                                        mon.die();
                                }
                                else
                                {
                                        this.die();
                                        break;
                                }
                        }
        }

}
```

4. Lớp Map

Nếu coi lập trình viên lập trình viên là thượng đế, thì bản đồ chính là một "thế giới" thu nhỏ nơi mà thượng đế cho vận hành những quy luật để tạo ra "cuộc sống". Mặc dù bản đồ có thể rất đơn giản trong nhiều game, nhưng đối với dạng game phiêu lưu này, nó đóng vai trò rất quan trọng (có thể hơn so với Monster). Một "thế giới" quá nhỏ bé sẽ khiến người chơi nhanh kết thúc vòng chơi và không thể khơi lên sự tò mò muốn khám phá của họ.

Khi một vật thể (chướng ngại vật), ta sẽ xóa vùng tương ứng trên buffer (ảnh bản đồ) và gán giá trị lại cho ô đó trong dữ liệu là mặc định (0):

```
function clearCell(left,top,col,row)
{
        data[col+row*COLS] = 0;
        context.save();
        context.globalCompositeOperation = "destination-out";
        context.fillStyle = "rgba(0,0,0,1)";
        context.fillRect(left,top,CELL_SIZE,CELL_SIZE);
        context.restore();
}
```

Để kiểm tra va chạm với các chướng ngại vật, ta tạo phương thức collide() với giá trị trả về là một đối tượng lưu giữ các giá trị về vị trí, kích thước của chướng ngại vật đó:

```
this.colllide = function(x,y,canDestroy){
        var b = this.contain(x,y);

        if(b)
        {
                if(canDestroy && b.type==BRICK)
                {
                        clearCell(b.left,b.top,b.col,b.row);
                }
                return b;
        }
        return false;
};
this.contain = function(x,y){
        var col = Math.floor(x/CELL_SIZE);
        var row = Math.floor(y/CELL_SIZE);
        var val = data[col+row*COLS];
        if(val>0)
        {
                var b = {
                        left: col*CELL_SIZE,
                        top: row*CELL_SIZE,
                        col: col,
                        row: row,
                        type: val,
                };
                b.right = b.left+CELL_SIZE;
                b.bottom = b.top+CELL_SIZE;

                return b;
        }
```

```
        return false;
};
```

Trong game này, lớp Map còn là nơi chứa các Monster. Để làm Monster ra liên tục nếu số lượng ít hơn một giá trị nào đó. Ta viết phương thức update() như sau:

```
this.update = function(){

        // generate random monsters
        if(this.monsters.length<MONSTER_IN_VIEW)
                this.monsters.push(new
Monster(this,this.offsetX+Math.random()*this.viewWidth+50,1));

        i = 0;
        var length = this.monsters.length;
        while(i<length){
                if(this.monsters[i].isDead                                    ||
this.monsters[i].left<this.offsetX)
                {
                        this.monsters.splice(i,1); // remove this monster from
array
                        length--;
                }else
                {
                        this.monsters[i].update();
                        i++;
                }
        }
};
```

H. Một số ứng dụng minh họa

Trong quá trình thực hiện bài báo cáo này, tôi thực hiện một vài dự án nhỏ để minh họa các kiến thức đã trình bày. Các dự án này hoàn toàn có thể được phát triển trở thành những game online để cung cấp cho người chơi trong thực tế.
Các dự án này được thực hiện bằng thuần HTML5 và không sử dụng thêm bất kì thư viện nào.

I. Game đua xe

1. Các thông số của xe

Trong game này, ta sẽ sử dụng bốn phím mũi tên để điều khiển xe. Hai phím UP, DOWN để di chuyển tiến lùi và LEFT, RIGHT sẽ dùng để quay hướng xe. Một xe đua cơ bản cần các thuộc tính sau:

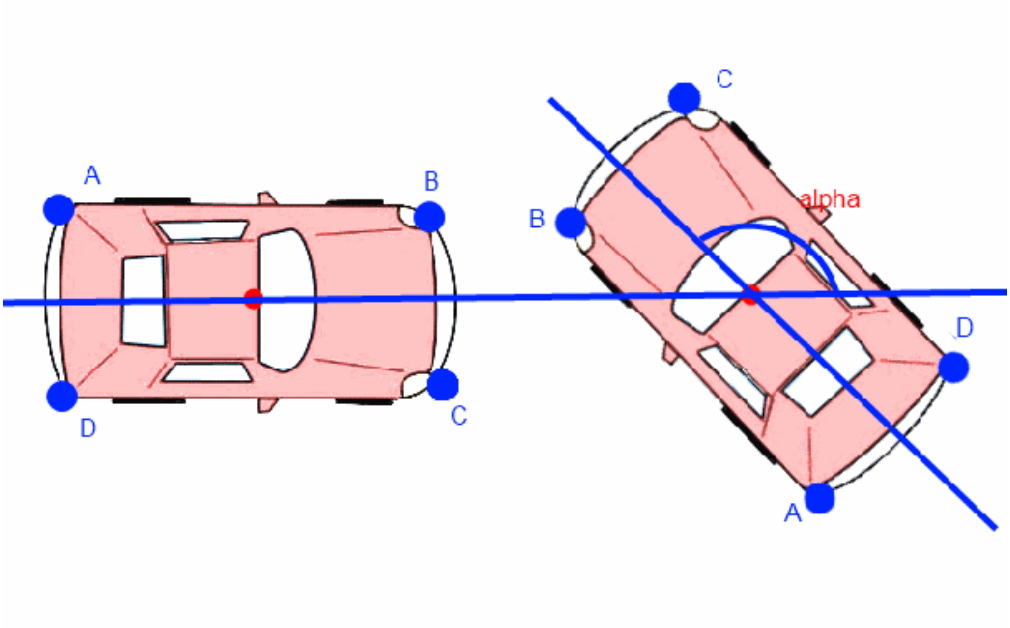
- max/min speed: tốc độ tối đa/tối thiểu của xe. Nên để tốc độ tối đa có giá trị tuyệt đối cao hơn tốc độ tối thiểu.

- acceleration: Khả năng tăng tốc của xe. Giá trị càng lớn thì xe càng nhanh đạt vận tốc tối đa.
- rotationAngle: Khả năng điều chỉnh góc quay của xe.
- friction: độ ma sát của xe trên từng loại địa hình. Giá trị này sẽ giúp xe giảm tốc độ khi người chơi không giữ phím di chuyển.

2. Di chuyển và quay xe

Với mục đích kiểm tra va chạm, ta sẽ sử dụng một mảng các điểm bao quanh xe hay xảy ra tiếp xúc nhất. Ở đây, các điểm mà ta chọn là 4 đỉnh từ vùng bao hình chữ nhật của xe (có thể coi như 4 bánh xe). Khi xe xoay góc alpha và di chuyển, ta phải tính lại tọa độ các điểm này tương ứng. Công thức để tính tọa độ mới của một điểm sau khi xoay góc alpha là:

x’ = cos(alpha) * x – sin(alpha) * y
y’ = sin(alpha) * x + cos(alpha) * y



Ví dụ tại góc 0 độ, với gốc tọa độ nằm ở tâm hình chữ nhật, điểm A có tọa độ là {x: -width/2,y: -height/2}. Vậy với góc xoay alpha, tọa độ mới của A là:

x: cos(alpha)*(-width/2) – sin(alpha)*(-height/2)
y: sin(alpha)*(-width/2) + cos(alpha)*(-height/2)

3. Kiểm tra va chạm (tiếp xúc) với địa hình

Với bản đồ là một hình ảnh, ta cần kiểm tra bằng cách dựa vào pixel. Tuy nhiên không phải bằng cách lặp mà dựa vào kiểm tra một vài vị trí xác định. Các vị trí này ta chính là 4 điểm thuộc các góc của xe mà ta đã xác định trong phần trước.

- Đầu tiên ta cần lấy đối tượng ImageData của ảnh làm bản đồ.

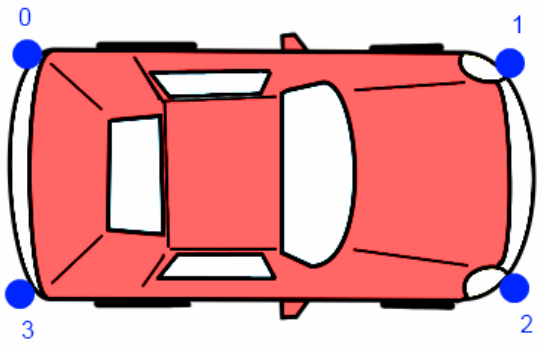
- Với mỗi điểm dùng để kiểm tra va chạm của xe, ta tính vị trí alpha của chúng trong ImageData ((x+y*width)*4+3) và so sánh giá trị alpha với 0 (tương ứng với mặt đường). Thay vì kiểm tra giá trị alpha, ta có thể kiểm tra màu sắc tại pixel đó cho từng loại địa hình khác nhau.

- Tăng, giảm ma sát ứng với từng loại địa hình.

Trong ví dụ này, ta chỉ tạo bản đồ với hai loại địa hình là đường và bãi cỏ. Độ ma sát sẽ tăng dần theo số lượng điểm (bánh xe) tiếp xúc với cỏ:

4. Hạn chế xe di chuyển và xoay khi bị va chạm

Sau khi biết được một điểm va chạm với đá, ta sẽ xác định xem xe có thể di chuyển hoặc xoay được hay không. Xem hình vẽ sau, ta xét từng trường hợp điểm va chạm của xe là:
- 0 hoặc 3: không cho phép xe lùi.
- 1 hoặc 2: không cho phép xe tiến.
- 0 hoặc 1: không cho phép xoay trái.
- 2 hoặc 3: không cho phép xoay phải.



5. Tạo các checkpoint

Để biết xe có đi đúng đường và đúng hướng, ta cần tạo ra các điểm kiểm tra trên đường gọi là checkpoint. Nếu người chơi muốn "đi tắt" đến checkpoint, hãy đảm bảo rằng họ sẽ đến chậm hơn so với đi trên đường chính bằng cách tạo các vật cản hoặc tăng ma sát.

Với bản đồ trong game này, ta chỉ tạo ra 4 checkpoint. Phương thức reachNextCheckPoint() sẽ kiểm tra xem xe có chạm checkpoint tiếp theo hay không. Khi đến checkpoint cuối cùng, ta sẽ đưa biến currentCheckPoint về 0 và tăng lap lên 1.

6. Kết quả

Online Demo.

II. Game bắn đại bác

1. Bản đồ và địa hình

Đối với dạng game này, địa hình của bản đồ có thể ảnh hưởng rất lớn đến người chơi. Ví dụ người chơi có thể rơi xuống một hố sâu và không thể bắn hay thậm chí "thiệt mạng". Tuy nhiên ở phần 1 này ta chưa cần quan tâm đến những vấn đề này. Phần chính mà ta hướng dẫn là làm sao để người chơi có thể tương tác và chịu tác động của địa hình như di chuyển, bắn phá.

Về vấn đề kiểm tra va chạm với địa hình, không có phương pháp nào khác ngoài việc kiểm tra dựa trên pixel (do địa hình có hình dạng phức tạp và bất kì). Vì vậy ta tạo một ImageData từ ảnh bản đồ, sau đó thêm một phương thức kiểm tra một điểm có nằm trong vùng ImageData có độ alpha bằng 0 hay không.

```
this.contain = function(x,y){
        if(!imageData)
                return false;
        var index = Math.floor((x+y*width)*4+3);
        return imageData.data[index]!=0;
}
```

2. Phá hủy một phần địa hình

Sử dụng phương thức contain bên trên, ta sẽ kiểm tra được va chạm khi đạn bắn hoặc người chơi rơi xuống đất. Với trường hợp đạn bắn, ta phải phá hủy vùng địa hình nơi đạn bay vào. Rất may là API của Canvas cung cấp một phương pháp dùng để vẽ ra các vùng có độ alpha bằng 0, tương tự với việc xóa bỏ hoàn toàn vùng đó.

Ta thực hiện việc này bằng cách gán hai thuộc tính của context canvas là **globalCompositeOperation** thành "destination-out" và fillStyle thành "rgba(0,0,0,1)". Và tại vị trí bị đạn bắn, ta sẽ vẽ một hình tròn để "khoan" vùng địa hình này. Bạn nên lưu và phục hồi lại context khi sử dụng thiết lập này.

3. Trọng lực và Gió

Trong dạng game này, đạn khi được bắn ra sẽ chịu tác động cùng lúc của 3 loại lực là: lực bắn, trọng lực và gió. Mỗi lực này có thể được minh họa bởi một vector (Lực bắn đã được giới thiệu trong phần trước). Tại mỗi thời điểm khi đạn bay trong không gian, vị trí mới của nó sẽ được tính bằng cách dùng vị trí hiện tại cộng với vector tổng của 3 lực này.

Lưu ý: Bạn sẽ thấy rằng tốc độ đạn bay quá nhanh có thể khiến cho việc kiểm tra va chạm không chính xác (do trong game là đạn "biến" từ nơi này đến nơi khác). Vì vậy cần giảm tốc độ đạn lại và tăng FPS lên một giá trị thích hợp. Để tạo gió, ta sử dụng một vector với hai giá trị x,y thay đổi ngẫu nhiên sau một khoảng thời gian khoảng 20 giây.

4. Di chuyển Cannon

Do có địa hình phức tạp, việc di chuyển cannon sẽ tương đối phức tạp và còn tùy thuộc vào hình dạng của nó. Việc cần quan tâm ở đây là cách để cannon có thể đi lên địa hình dốc vừa phải. Một mẹo nhỏ mà ta làm ở đây là sử dụng cách "nhảy cóc" thay vì đi. Như vậy mỗi bước đi theo chiều ngang ta cũng đồng thời nâng cao vị trí của cannon hơn một chút. Nếu cannon đi xuống dốc, nó sẽ rơi xuống vị trí thấp hơn (nhờ phương thức update()); nếu cannon đi lên dốc, nó sẽ ở vị trí cao hơn. Đối với dốc có độ nghiêng lớn, ta sẽ kiểm tra một tọa độ bên trái (hoặc phải tùy theo hướng di chuyển) xem nó có chạm vào mặt địa hình không. Nếu có, ta sẽ không cho phép cannon di chuyển theo hướng đó.

5. Sát thương của đạn

Mỗi khi đạn trúng đất hoặc bất kì cannon nào, nó sẽ phát nổ mà gây sát thương cho các cannon nằm trong phạm vi nhất định. Sức sát thương của đạn còn bị ảnh hưởng bởi tốc độ của nó khi

trúng mục tiêu. Vậy ta tạo một thuộc tính lưu trữ năng lượng của viên đạn khi đang bay dựa vào công thức động lượng (E = 1/2*m*v^2).

Lượng hp bị tổn thất của cannon khi trúng đạn sẽ được tính bằng tổng của động năng viên đạn và sát thương khi phát nổ. Lực khi phát nổ ta sẽ cho giá trị từ 0 đến 100 với mọi phạm vi sát thương khác nhau của đạn. Với cách tính này, bạn có thể đảm bảo rằng khi sử dụng một loại đạn có phạm vi sát thương lớn thì mức độ sát thương của nó cũng chỉ tương đương với loại có phạm vi sát thương nhỏ hơn (cần phân biệt lực và phạm vi sát thương của đạn).

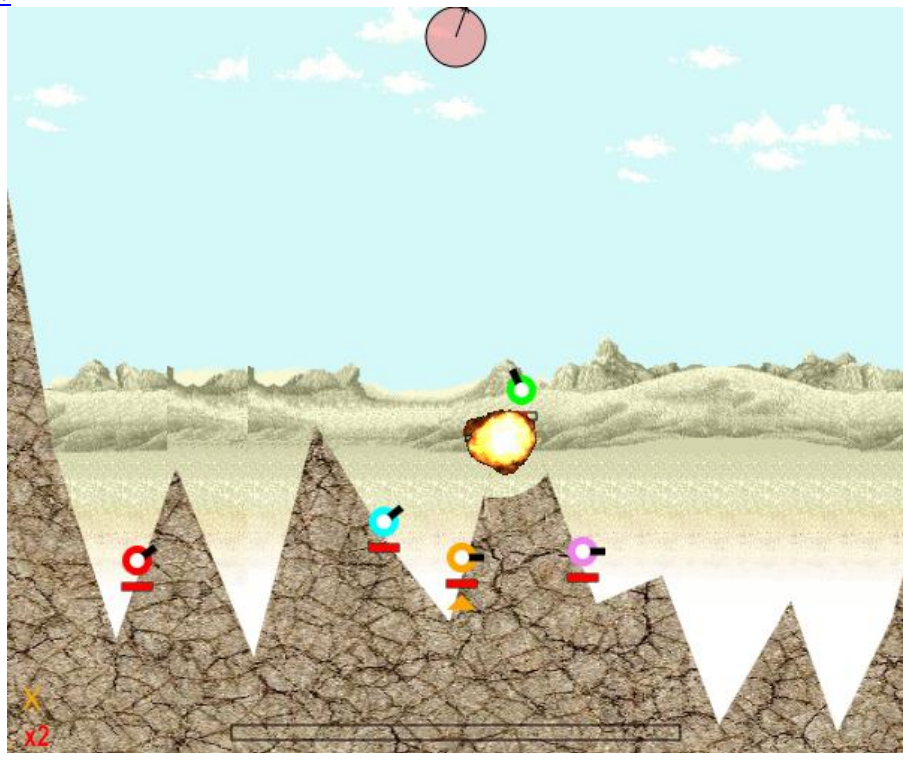
6. Hỗ trợ nhiều người chơi

Bởi vì game không chơi qua mạng nên nhiều người phải chia sẻ cùng một màn hình. Mỗi khi đạn phát nổ hoặc bay ra khỏi bản đồ, ta sẽ thực hiện chuyển lượt chơi đến người tiếp theo. Nếu như người chơi tiếp theo đã chết, tiếp tục gọi đệ quy với điều kiện dừng là số người chơi còn sống nhỏ hơn 1 (_alivePlayers<1).

Ngoài nhiệm vụ chính trên, phương thức changeTurn() dưới đây còn có nhiệm vụ hiển thị hiệu ứng nổ tại vị trí của viên đạn, bởi vì đây là thời điểm ngay sau khi nó phát nổ.

7. Kết quả

Online Demo.



III. Game Mario

Đây là kết quả của việc phát triển "Một nền tảng game 2D side-scrollong" từ phần trước.

Online Demo.

I. Lời kết

Hiện tại đã có rất nhiều sản phẩm game cũng như các ứng dụng mang tính đồ họa tương tác cao được làm trên nền tảng HTML5. Các thiết bị di động và hệ điều hành mới như Windows 8 cũng tập trung vào hỗ trợ và sử dụng công nghệ HTML5. Kỉ nguyên của HTML5 có thể mở ra một tiêu chuẩn thống nhất cho các ứng dụng rich user experience và không cần phải đắn đo trong việc lựa chọn các thư viện hỗ trợ.
Các nội dung trình bày trong sách này chưa thật sự nâng cao và tập trung vào việc phát triển các dự án thành một sản phẩm game thực sự có thể cạnh tranh trên thị trường. Tuy nhiên đây là bước khởi đầu vững chắc cho tương lai của việc phát triển game trên mạng với công nghệ HTML5. Kết hợp với mô hình client-server, ta có thể phát triển các game nhiều người chơi (MMO) và lưu trữ dữ liệu trực tiếp của người chơi trên server. Đây là điều mà tôi chưa đề cập tới trong báo cáo này.

J. Tài liệu tham khảo

1. HTML 5 Tutorial (http://html5tutorial.net/).
2. Dive Into HTML 5 (http://diveintohtml5.info/offline.html).
3. HTML 5 Specification (http://www.w3.org/TR/2011/WD-html5-20110525/).
4. HTML 5 Rocks (http://www.html5rocks.com/en/).
5. HTML 5 Doctor (http://html5doctor.com/)
6. HTML 5 Demos and Examples (http://html5demos.com/)
7. Mozilla Development Netword – HTML 5 (https://developer.mozilla.org/en/html/html5/)
8. Google.com.