

# Développer des jeux en **HTML5 & JavaScript**

Multijoueur temps-réel avec Node.js et intégration dans Facebook

**Samuel Ronce**



# Développer des jeux en HTML5 & JavaScript

Alors que les navigateurs sont en constante évolution pour optimiser l'exécution de jeux, HTML5 propose de nouvelles balises et API JavaScript pour gérer la vidéo et l'audio, dessiner en 2D et 3D. De quoi alimenter l'explosion du marché des jeux web, natifs et sociaux, notamment sur les smartphones et tablettes.

## De la conception du gameplay à la configuration du serveur Node.js et MongoDB

Cet ouvrage, exemples de code à l'appui, décortique les étapes de création d'un jeu vidéo en HTML5 et JavaScript, de la réalisation d'un concept et de l'étude de la concurrence, à la commercialisation et à l'animation d'une communauté. Tous les aspects du développement sont abordés : conception du scénario et du gameplay, création des niveaux (*level design*) et des profils de joueurs, gestion de spritesheets, mise au point d'une ambiance, affichage de décors, effets graphiques, animations, collisions et effets sonores (HTML5 Audio)...

L'ouvrage guide également le lecteur dans le choix d'un framework, et la configuration d'un serveur pour du temps réel avec MongoDB et Node.js. Il explique enfin comment intégrer le jeu dans le réseau social Facebook.

## Au sommaire

Mettre au point le concept du jeu • Action • Jeu de rôle • Aventure • Réflexion • Choisir un framework • Installer CanvasEngine • Créer l'écran titre • La scène • La barre de progression • Les boutons • Afficher les décors • Utiliser Tiled Map Editor • Les sprites • Animer les éléments sur l'écran • Animation en boucle • Réaction à une action • Concevoir le Gameplay • Plate-forme mobile • Contrôle du joueur avec un clavier, une manette • Accéléromètre • Accélération et décélération • Gravité • Saut • Mouvoir le joueur avec des défilements • Déplacements • Rafraîchissement • Interaction avec les objets • Mise en place des règles du jeu • Affichage du score • Ambiance • HTML5 Audio • SoundManager pour les effets sonores • Effet jour/nuit • Les adversaires • Affichage des dommages • Champ de vision • Réaliser la sauvegarde • Crée un jeu plate-forme • Initialisation et création des classes • Chargement du niveau • Création des animations • Défillement de la carte • Gestion des collisions • Mouvement • Configurer le serveur avec Node.js pour le multijoueur en temps réel • Utiliser le serveur via SSH • Télécharger Node.js. Installer NPM • MongoDB et Mongoose • Intégration du jeu au réseau social Facebook • Authentification et autorisation • Implémentation de la partie Social Gaming • Inviter des amis à jouer • Afficher un score et le partager sur le mur de l'utilisateur • Système de badges • Récupérer des informations pour les utiliser dans le jeu • Stratégie de monétisation • Monnaie virtuelle • Annexes • Rappels sur HTML5 Canvas • Dessiner • Lignes • Arcs • Chemins • Formes • Afficher un texte • Des couleurs • Des ombres • Frameworks JavaScript • Easel.js • RPG JS • Crée des jeux de rôle • 3D avec Three.js • Crée une scène • Ajouter un objet • Des sources de lumière • Rendu • Bouger la caméra avec la souris.

## À qui s'adresse cet ouvrage ?

- Aux développeurs web, amateurs ou professionnels, mais initiés au langage JavaScript, qui souhaitent se lancer dans la création de jeux en HTML5 ;
- Aux agences web et de communication souhaitant se renseigner sur le potentiel des jeux communautaires.



Samuel Ronce

Spécialisé dans la création d'applications web et de jeux vidéo en JavaScript et HTML5, il est le fondateur de WebCreative5, société qui développe des moteurs de jeux vidéo Open Source en pur HTML5, compatibles avec les dernières technologies.

Développer des  
**jeux**  
en **HTML5 & JavaScript**

---

DANS LA MÊME COLLECTION

R. RIMELÉ. – **HTML 5. Une référence pour le développeur web.**  
N°13638, à paraître en 2013, 644 pages (collection Blanche).

J. STARK. – **Applications iPhone avec HTML, CSS et JavaScript. Conversions en natifs avec PhoneGap.**  
N°12745, 2010, 190 pages (collection Blanche).

R. GOETTER. – **CSS avancées. Vers HTML 5 et CSS 3.**  
N°13405, 2<sup>e</sup> édition, 2012, 400 pages (collection Blanche).

E. SARRION. – **jQuery Mobile. La bibliothèque JavaScript pour le Web mobile.**  
N°13388, 2012, 610 pages.

J.-M. DEFRENCE. – **Ajax, jQuery et PHP. 42 ateliers pour concevoir des applications web 2.0.**  
N°13271, 3<sup>e</sup> édition, 2011, 482 pages (collection Blanche).

C. PORTENEUVE. – **Bien développer pour le Web 2.0. Bonnes pratiques Ajax.**  
N°12391, 2<sup>e</sup> édition, 2008, 674 pages (collection Blanche).

F. DAOUST, D. HAZAËL-MASSIEUX. – **Relever le défi du Web mobile. Bonnes pratiques de conception et de développement.**  
N°12828, 2011, 300 pages (collection Blanche).

E. DASPET, C. PIERRE DE GEYER. – **PHP 5 avancé.**  
N°13435, 6<sup>e</sup> édition, 2012, 900 pages environ (collection Blanche).

J. PAULI, G. PLESSIS, C. PIERRE DE GEYER. – **Audit et optimisation LAMP.**  
N°12800, 2012, 300 pages environ (collection Blanche).

P. BORGHINO, O. DASINI, A. GADAL. – **Audit et optimisation MySQL 5.**  
N°12634, 2010, 282 pages (collection Blanche).

S. JABER. – **Programmation GWT 2.5. Développer des applications HTML5/JavaScript en Java avec Google Web Toolkit.**  
N°13478, 2<sup>e</sup> édition, 2012, 540 pages (collection Blanche).

---

CHEZ LE MÊME ÉDITEUR

J. ENGELS. – **HTML5 et CSS3. Cours et exercices corrigés.**  
N°13400, 2012, 550 pages.

E. SARRION. – **jQuery 1.7 & jQuery UI.**  
N°13504, 2<sup>e</sup> édition, 2012, 600 pages.

E. SARRION. – **Mémento jQuery.**  
N°13488, 2012, 14 pages.

K. DELOUMEAU-PRIGENT. – **CSS maintenables avec Sass & Compass. Outils et bonnes pratiques pour l'intégrateur web.**  
N°13417, 2012, 272 pages (collection Design web).

C. SCHILLINGER. – **Intégration web : les bonnes pratiques. Le guide du bon intégrateur.**  
N°13370, 2012, 400 pages (collection Design web).

I. CANIVET et J-M. HARDY. – **La stratégie de contenu en pratique. 30 outils passés au crible.**  
N°13510, 2012, 176 pages (collection Design web).

S. DAUMAL. – **Design d'expérience utilisateur. Principes et méthodes UX.**  
N°13456, 2012, 208 pages (collection Design web).

E. SARRION. – **Prototype et Scriptaculous. Dynamiser ses sites web avec JavaScript.**  
N°85408, 2010, 342 pages (e-book).

G. SWINNEN. – **Apprendre à programmer avec Python 3.**  
N°13434, 3<sup>e</sup> édition, 2012, 435 pages.

J. ENGELS. – **PHP 5. Cours et exercices.**  
N°12486, 2009, 638 pages.

P. ALEXIS et H. BERSINI. – **Apprendre la programmation web avec Python et Django. Principes et bonnes pratiques pour les sites web dynamiques.**  
N°13499, 2012, 344 pages (collection Noire).

H. BERSINI. – **La programmation orientée objet.**  
N°12806, 5<sup>e</sup> édition, 2011, 644 pages.

C. SOUTOU. – **Programmer avec MySQL.**  
N°12869, 2<sup>e</sup> édition, 2011, 450 pages.

T. BAILLET. – **Créer son propre thème WordPress pour mobile.**  
N°13441, 2012, 128 pages (collection Accès libre).

E. SLOÏM. – **Mémento Sites web. Les bonnes pratiques.**  
N°12802, 3<sup>e</sup> édition, 2010, 18 pages.

A. BOUCHER. – **Ergonomie web illustrée. 60 sites à la loupe.**  
N°12695, 2010, 302 pages. (Design & Interface).

I. CANIVET. – **Bien rédiger pour le Web. Stratégie de contenu pour améliorer son référencement naturel.**  
N°12883, 2<sup>e</sup> édition, 2011, 552 pages.

N. CHU. – **Réussir un projet de site web.**  
N°12742, 6<sup>e</sup> édition, 2010, 256 pages.

H. COCIAMONT. – **Réussir son premier site Joomla! 2.5.**  
N°13425, 2012, 160 pages.

S. BORDAGE, D. THÉVENON, L. DUPAQUIER, F. BROUSSE. – **Conduite de projet Web.**  
N°13308, 6<sup>e</sup> édition, 2011, 480 pages.

# Développer des jeux en **HTML5 & JavaScript**

Multijoueur temps-réel avec Node.js et intégration dans Facebook

**Samuel Ronce**

ÉDITIONS EYROLLES  
61, bd Saint-Germain  
75240 Paris Cedex 05  
[www.editions-eyrolles.com](http://www.editions-eyrolles.com)

Remerciements à Romain Pouclet pour sa relecture ainsi qu'à Anne Bougnoux

# Avant-propos

---

## Pourquoi concevoir un jeu en HTML5/JavaScript ?

Depuis des années, le langage Javascript était utilisé avec parcimonie pour quelques interactions avec l'utilisateur. Toutefois, les nouvelles versions de navigateurs web ont accéléré l'interprétation du Javascript, autorisant la création d'applications plus sophistiquées, et particulièrement de jeux. Et surtout, HTML5 a fait son apparition ; il propose de nouvelles balises et API pour JavaScript, afin de manier notamment la vidéo et l'audio, ainsi que la possibilité de dessiner en 2D ou 3D.

### COMPATIBILITÉ Les balises de dessin

Les balises de dessin (`<canvas>`) existaient déjà dans Safari en 2004. Cependant, elles étaient très peu exploitées ; car, d'une part, les autres navigateurs ne les utilisaient pas et, d'autre part, les navigateurs n'étaient pas assez puissants pour réaliser des jeux. La technologie Flash était encore bien présente.

HTML5 permet de dynamiser des pages web comme Flash le faisait antérieurement. Alors pourquoi ne pas rester sur des développements Flash, accessibles pour 99 % des utilisateurs, quand environ 40 % de ces derniers comprennent HTML5 ? La question se pose moins en termes d'accessibilité qu'en termes d'évolution : développer maintenant en HTML5, c'est anticiper sur la mise en place de produits sur de nouveaux marchés. Il suffit de constater l'explosion du marché des Smartphones – 1 milliard dans le monde prévu en 2013 – et des tablettes tactiles. L'intérêt de HTML5 réside dans sa disponibilité pour de nombreuses plates-formes. Votre jeu fonctionnera aussi bien sur iOS ou Android que sur les télévisions connectées.

**CONCURRENCE Que devient Flash ?**

Adobe a abandonné Flash Player pour les mobiles et déclaré sur son blog : « *HTML5 est désormais pris en charge par les principaux appareils mobiles, dans certains cas exclusivement. Cela en fait la meilleure solution pour créer et diffuser des contenus dans le navigateur sur différentes plates-formes mobiles* ».

Adobe a tout de même tenté d'implanter sa technologie AIR qui permet de « cross-compiler » vers ces nouvelles plates-formes. AIR étant sans doute plus mature que HTML5 aujourd'hui, il pourrait être envisagé de continuer avec *Actionscript*. Néanmoins, si le jeu est porté sur navigateur et essayé sur iPad, comment le joueur peut-il l'essayer sans HTML 5 ?

**REMARQUE Jeu natif**

Il est possible de faire tourner un jeu nativement en utilisant PhoneGap ou Adobe AIR 3.

Certes, HTML5 est encore jeune et en cours de standardisation, mais cela n'empêche pas des facteurs importants du Web d'en faire la promotion dès maintenant, voire de l'utiliser. Par exemple, il est possible de développer des applications en HTML5 sur Windows 8 UI de Microsoft. Inutile d'hésiter, utiliser HTML5 n'est pas une décision délicate pour une entreprise. Toutefois, selon le public visé, pensez à adapter votre application pour des navigateurs plus anciens tels que Internet Explorer 7 et 8.

**ANCIENNES VERSIONS La rétro-compatibilité pour des jeux**

La balise `<canvas>` pour les jeux HTML5 n'existe pas du tout sur la plupart des anciens navigateurs. ExplorerCanvas est un code JavaScript obligeant IE8 à interpréter cette balise.

Il est conseillé de demander à l'internaute de mettre à jour son navigateur, pour que son expérience du jeu soit plus agréable.

► <http://code.google.com/p/explorercanvas>

Devant l'enthousiasme général soulevé par HTML5, de nombreux projets d'adaptation ont été pratiquement abandonnés : dernier *commit* en 2010 pour ExplorerCanvas, dernière *release* de FlashCanvas en 2011.

Créer un jeu en HTML5, est-ce possible ? On est loin de la petite interaction de la part de l'utilisateur. Non seulement des algorithmes doivent être codés, ce qui implique une interprétation plus ardue du JavaScript, mais l'affichage graphique est également plus poussé avec des rafraîchissements continuels. Heureusement, les navigateurs web ont été largement améliorés et donnent désormais au développeur la possibilité de concevoir un jeu en HTML5.

Cependant, cela n'exclut pas les limites de l'interprétation et ne remplace pas la puissance de langages comme C ou C++, plus adaptés pour des jeux de très grande qualité que l'on retrouve sur des consoles de salon. Restons-en pour l'instant à la 2D en HTML5 ; programmer en 3D est actuellement prématué, mais le jour viendra... Bien entendu, cela ne vous empêche pas de travailler sur la 3D avec WebGL.

**APPROFONDIR WebGL et Three.js**

La dernière annexe de l'ouvrage présente succinctement la conception d'éléments 3D avec Three.js.

Réaliser un jeu en 2D, n'est-ce pas un problème stratégique si l'on souhaite attaquer le marché des jeux vidéo ? Cela dépend de votre positionnement. A priori, si vous créez un jeu en HTML5, il sera distribué en ligne, ainsi que sur les nouvelles technologies (smartphones et tablettes tactiles). En ligne, soit vous créez votre propre site contenant votre jeu, soit vous utilisez les réseaux sociaux. On ne peut que conseiller la deuxième option (ou les deux) puisque les réseaux sociaux, tels que Facebook ou Google+, ont des millions d'utilisateurs qui sont autant de joueurs occasionnels ne cherchant pas la 3D HD mais l'amusement temporaire, seuls ou entre joueurs.

**STATISTIQUES Qui sont les joueurs des réseaux sociaux ?**

69 % des joueurs sont des femmes et ont en moyenne 43 ans (PopCap Games 2011). Ces statistiques sont loin des clichés sur les joueurs adolescents.

Un public varié, des plates-formes différentes... un jeu 2D, s'il est bien conçu, n'aura aucun mal à se placer dans ce marché, même en pleine génération 3D HD. Prenons des exemples concrets :

- Angry Birds, réalisé par Rovio, a engendré plus de 50 millions d'euros de chiffre d'affaires pour un budget d'environ 100 000 euros. Pourtant, le concept du jeu est très simple et la 3D n'est pas présente.
- FarmVille de Zinga, simulateur de vie agricole, est une application très populaire sur Facebook avec plus de 82 millions d'utilisateurs actifs.

Bien sûr, les échecs sont également nombreux dans le monde des jeux, mais c'est moins en raison de la 2D que pour des critères de « jouabilité » : sans mobiliser un budget conséquent, il suffit d'éditer un jeu addictif et amusant, pour lequel la compétitivité entre amis (faire le meilleur score, la plus grosse ville, etc.) devient très intéressante quand les réseaux sociaux s'en mêlent.

## À qui s'adresse cet ouvrage ?

Que vous soyez amateur ou professionnel, cet ouvrage vous fournira les bases pour la création d'un jeu en HTML5. Toutefois, il s'adresse aux personnes connaissant le langage JavaScript. Si vous souhaitez attaquer le marché des jeux vidéo, le livre complètera tout l'aspect technique par un chapitre sur les stratégies de monétisation pour récompenser vos efforts.

## Structure du livre

Le livre s'organise en trois parties dans un ordre logique :

- 1 la réalisation d'un concept de jeu;
- 2 le développement technique;
- 3 l'intégration dans les réseaux sociaux.

Le concept est une réflexion préalable sur la réalisation du jeu. Le chapitre 1 vous aide dans votre étude du marché et à vous diriger vers un contexte de jeu (Gameplay) attrayant.

Le chapitre 2 mentionne le chargement des ressources graphiques et audio avant le jeu et les différents écrans qui donneront la première impression au joueur.

Le chapitre 3 vous apprend à construire une structure pour l'importation des données dans le jeu afin de réaliser une carte ou un niveau et de s'aider d'un éditeur OpenSource.

Le chapitre 4 explique comment concevoir des animations à partir d'une image, ou déformer un élément selon une frise temporelle virtuelle.

Le chapitre 5 expose la réalisation du Gameplay avec le clavier, la souris, l'accéléromètre ou la manette. Il enseigne comment diriger un personnage en se souciant de son accélération, sa gravité, etc.

Le chapitre 6 détaille les méthodes de défilement de la carte pour pouvoir se déplacer dans l'intégralité du décor.

Le chapitre 7 montre comment créer les interactions entre le joueur et les décors pour gérer les collisions ainsi que les interactions entre le joueur et les autres personnages pour déclencher des événements.

Le chapitre 8 aide à mettre en place le concept avec l'application des règles du jeu.

Le chapitre 9 vous apprend à amplifier l'ambiance du jeu par des musiques et effets sonores ainsi qu'avec des effets graphiques.

Le chapitre 10 explique la conception des adversaires et le calcul des dégâts lors d'une attaque entre le joueur et l'adversaire.

Le chapitre 11 retrace les moyens pour sauvegarder et charger les données du jeu.

Le chapitre 12 résume les chapitres précédents par un cas pratique : la création d'un jeu plate-forme.

Le chapitre 13 mentionne l'installation de Node.js pour réaliser un jeu multijoueur.

Le chapitre 14 explique la création des modèles côté serveur pour le partage des données en temps réel.

Le chapitre 15 montre comment intégrer le jeu sur le réseau social Facebook avec authentification et autorisation du jeu.

Le chapitre 16 détaille la récupération des informations du joueur pour les partager avec ses amis et les inviter à leur tour de jouer.

Le chapitre 17 explique comment utiliser la monnaie virtuelle de Facebook comme stratégie de monétisation.

L'annexe *Rappels sur HTML5 Canvas* traite des différentes méthodes de l'élément `canvas` de HTML, utiles dans le dessin et l'affichage dans le jeu. Nous vous conseillons de la lire attentivement si vous n'êtes pas encore familiarisé avec cette balise.

Il existe plusieurs frameworks pour réaliser des jeux en HTML5. L'annexe *Frameworks Javascript* en mentionne deux : Easel.js et RPG.js.

La dernière annexe évoque la création de jeux 3D en WebGL avec l'aide du framework Three.js.

## Remerciements

Je remercie :

- l'équipe des éditions Eyrolles pour la publication de cet ouvrage, et particulièrement Muriel Shan Sei Fan pour le suivi de l'écriture ;
- David Dany pour quelques illustrations tirées d'un jeu coproduit ensemble ;
- Romain Pouclet pour la relecture technique ;
- Anne Bougnoux et Laurène Gibaud pour la relecture générale.



# Table des matières

---

<b>Avant-propos .....</b>	V
Pourquoi concevoir un jeu en HTML5/JavaScript ? .....	V
À qui s'adresse cet ouvrage ? .....	VII
Structure du livre .....	VIII
Remerciements .....	IX
CHAPITRE 1	
<b>Mettre au point le concept du jeu .....</b>	1
<b>Étude de marché .....</b>	2
Quelques géants du Social Gaming .....	2
Étude démographique et comportementale .....	3
Étude technologique .....	5
Votre étude de marché .....	6
<b>Positionnement .....</b>	6
Joueurs ciblés .....	6
Prix du jeu .....	8
<b>Type du jeu .....</b>	8
Action .....	9
Jeu de rôle .....	9
Aventure .....	10
Action-Aventure .....	10
Simulation .....	10
Sport .....	11
Réflexion .....	11
<b>Choisir un framework .....</b>	11

Easel.js .....	11
Kinetic.js .....	12
Crafty.js .....	13
Caat.js .....	13
CanvasEngine.js .....	13
<b>Installer CanvasEngine .....</b>	<b>14</b>
 CHAPITRE 2	
<b>Créer l'écran titre .....</b>	<b>15</b>
<b>Qualité des images .....</b>	<b>16</b>
<b>Chargement initial .....</b>	<b>16</b>
Création de la scène de chargement .....	17
Schéma des données .....	17
Affichage de la barre de progression .....	18
<b>Écran titre .....</b>	<b>21</b>
Création de la scène de l'écran titre .....	21
Initialisation des boutons .....	22
Association des événements .....	24
<b>Écrans additionnels .....</b>	<b>25</b>
Options .....	25
Niveaux .....	26
 CHAPITRE 3	
<b>Affichage des décors .....</b>	<b>27</b>
<b>Level Design .....</b>	<b>28</b>
Insertion des données .....	28
Une carte .....	29
Un niveau .....	34
<b>Utiliser Tiled Map Editor .....</b>	<b>38</b>
Créer la carte .....	38
Intégrer la carte dans la scène .....	39
Obtenir des données de la carte .....	40
<b>Objets principaux : les sprites .....</b>	<b>40</b>
Ensembles d'éléments graphiques ou Spritesheets .....	41
Cas particulier .....	45

<b>CHAPITRE 4</b>	
<b>Animer les éléments sur l'écran .....</b>	47
<b>Déformer pour animer .....</b>	48
<b>Animation en boucle .....</b>	49
<b>Animer en réaction à une action .....</b>	52
<b>Animation temporaire .....</b>	53
<b>CHAPITRE 5</b>	
<b>Concevoir le Gameplay .....</b>	57
<b>Mouvement .....</b>	58
État d'un élément .....	58
Exemple : plate-forme mobile .....	58
<b>Contrôle du joueur .....</b>	61
Clavier .....	63
Souris .....	64
Accéléromètre .....	65
<b>Accélération et décélération .....</b>	71
Accélération .....	72
Décélération .....	73
<b>Gravité pour le saut .....</b>	75
Initialisation .....	76
Gravité .....	77
Saut .....	77
<b>CHAPITRE 6</b>	
<b>Avancer le joueur avec des défilements .....</b>	81
<b>Défilement classique .....</b>	82
Définir les éléments à déplacer .....	82
Rafraîchissement du déplacement .....	83
<b>Défilement différentiel .....</b>	83
<b>CHAPITRE 7</b>	
<b>Interaction avec les objets .....</b>	85
<b>Collision .....</b>	86

Au bord de la carte .....	87
Sur le décor et les objets .....	89
Collision sur des objets .....	93
<b>Interaction .....</b>	<b>96</b>
Déclenchement automatique au contact .....	96
Selon une action .....	99
<b>CHAPITRE 8</b>	
<b>Mise en place des règles du jeu .....</b>	<b>101</b>
<b>Situation initiale du joueur .....</b>	<b>102</b>
Inventaire .....	103
<b>Application du concept .....</b>	<b>104</b>
Affichage des points de vie .....	104
Explication des règles au joueur .....	106
<b>Affichage du score .....</b>	<b>108</b>
<b>Fin de partie : gagnée ou perdue .....</b>	<b>109</b>
Le joueur termine un parcours .....	109
Le joueur sort de l'écran vers le bas .....	110
<b>CHAPITRE 9</b>	
<b>Ambiance du jeu .....</b>	<b>111</b>
<b>Ajouter des effets sonores .....</b>	<b>112</b>
HTML5 Audio .....	112
SoundManager .....	113
Effectuer des fondus musicaux .....	114
<b>Dynamiser avec des effets graphiques .....</b>	<b>115</b>
Ton de l'écran : effet jour/nuit .....	115
Flash visuel .....	115
<b>CHAPITRE 10</b>	
<b>Les adversaires .....</b>	<b>117</b>
<b>Paramètres des adversaires .....</b>	<b>118</b>
Modèle et affichage des points de vie .....	118
Calcul des dégâts .....	121
Zones spécifiques d'interaction .....	124

<b>Champ de vision</b> .....	127
Zone de détection .....	127
Réaction .....	128
CHAPITRE 11	
<b>Réaliser la sauvegarde</b> .....	131
Sérialisation des classes .....	132
Chargement des données .....	133
CHAPITRE 12	
<b>Cas pratique : créer un jeu plate-forme</b> .....	135
Règles du jeu .....	136
Initialisation et création des classes .....	136
Chargement du niveau .....	138
Création des animations .....	139
Défilement de la carte .....	140
Gestion des collisions .....	141
Mouvement, gravité et saut .....	143
Effectuer un mouvement selon une touche .....	145
CHAPITRE 13	
<b>Configurer le serveur pour le multijoueur</b> .....	149
Utiliser le serveur via SSH .....	150
Télécharger Node.js .....	151
Installer NPM .....	152
Installer Socket.io .....	152
Tester l'installation .....	153
CHAPITRE 14	
<b>Utilisez Node.js pour votre jeu multijoueur en temps réel</b> .....	155
Comment fonctionne Socket.io ? .....	156
Fonctionnement dans CanvasEngine .....	157
Définir les événements .....	157

Composer la structure du jeu .....	158
<b>Créer des modules dans Node.js .....</b>	<b>159</b>
Créer un modèle .....	159
<b>Base de données .....</b>	<b>162</b>
Schéma .....	162
Données dans le modèle .....	163
<b>Gérer les connexions et déconnexions .....</b>	<b>164</b>
<b>Données communes .....</b>	<b>165</b>
Partage des données entre joueurs .....	166
<b>Sauvegarde et chargement avec Mongoose .....</b>	<b>168</b>
Installation de MongoDB et Mongoose .....	168
Connexion à la base et schéma .....	169
Sauvegarder des données .....	169
Chargement .....	170
 CHAPITRE 15	
<b>Intégration du jeu à un réseau social : Facebook .....</b>	<b>171</b>
Déclaration du jeu dans Facebook .....	172
Authentification et autorisation .....	173
 CHAPITRE 16	
<b>Implémentation de la partie Social Gaming .....</b>	<b>177</b>
<b>Intégration du jeu HTML5 .....</b>	<b>178</b>
Intégration du SDK et initialisation .....	178
L'utilisateur est-il connecté ? .....	179
Scène pour demander une connexion .....	180
<b>Inviter des amis à jouer .....</b>	<b>181</b>
Supprimer la notification .....	182
<b>Afficher un score et le partager sur le mur de l'utilisateur .....</b>	<b>183</b>
<b>Système de badges .....</b>	<b>185</b>
<b>Récupérer des informations (amis, groupes, etc.) pour les utiliser dans le jeu .....</b>	<b>186</b>
Données générales .....	187
Liste des groupes .....	188

Liste des amis .....	188
<b>CHAPITRE 17</b>	
<b>Stratégie de monétisation .....</b>	<b>191</b>
<b>Monnaie virtuelle .....</b>	<b>192</b>
Pourquoi une monnaie virtuelle ? .....	192
Configuration et déploiement .....	193
<b>CHAPITRE A</b>	
<b>Rappels sur HTML5 Canvas .....</b>	<b>197</b>
<b>Initialiser et charger le canvas .....</b>	<b>198</b>
<b>Dessiner dans le canvas .....</b>	<b>199</b>
Les lignes .....	199
Les arcs .....	200
Chemins .....	201
Les formes .....	201
Recadrer .....	202
Les dégradés .....	203
<b>Afficher un texte .....</b>	<b>204</b>
Couleurs .....	205
<b>Les ombres .....</b>	<b>205</b>
<b>Composite .....</b>	<b>206</b>
<b>Insérer des images .....</b>	<b>207</b>
Redimensionner .....	208
Couper .....	208
Répéter l'image en fond .....	209
<b>Transformation .....</b>	<b>210</b>
Translation .....	210
Rotation .....	210
Redimensionnement .....	211
Transformation personnalisée .....	211
<b>Manipulation des pixels .....</b>	<b>212</b>

<b>CHAPITRE B</b>	
<b>Frameworks JavaScript</b>	215
<b>Easel.js</b>	216
Installer	216
Premiers pas	216
Ajouter des formes	216
Appeler le canvas en boucle	217
Afficher un texte	217
Ajouter des conteneurs	218
Insérer et animer une image	219
<b>RPG JS : créez des jeux de rôle</b>	220
Premiers pas	220
Transférer le joueur sur une autre carte	222
Créer un événement	222
Commandes événements	224
Ajouter dynamiquement des événements	225
Ajouter une animation	226
Créer des actions	229
<b>CHAPITRE C</b>	
<b>3D avec Three.js</b>	231
<b>Installation</b>	232
<b>Créer une scène</b>	232
<b>Ajouter un objet</b>	233
Former un groupe d'objets	234
<b>Source de lumière</b>	235
<b>Rendu</b>	235
<b>Bouger la caméra avec la souris</b>	236

# 1

## Mettre au point le concept du jeu

---

La création d'un jeu se base préalablement sur un concept réfléchi. Voici un tour d'horizon sur le marché actuel, le comportement des joueurs et les types des jeux.

## Étude de marché

L'émergence des ordinateurs « de poche » (smartphones, tablettes tactiles) et le développement gigantesque des réseaux sociaux ouvrent un nouveau marché.

Les personnes qui se connectent sur les réseaux sociaux ne le font pas dans le seul but de discuter ou partager des informations : 50 % des utilisateurs de Facebook se connectent seulement pour jouer (selon AllFacebook.com). Peut-on penser sérieusement que le *Social Gaming* est un marché financièrement intéressant ? En analysant le comportement des joueurs et le chiffre d'affaires des entreprises sur ce marché, nous pouvons répondre de manière positive sans hésitation. En 2016, le marché du Social Gaming est estimé à 11,3 milliards de dollars (selon le dernier rapport de l'industrie IBISWorld). De plus, le montant des dépenses sur les biens virtuels dans la première moitié de l'année 2012 était de 1,26 milliards de dollars (selon Forbes). Cette croissance s'explique par l'omniprésence des réseaux sociaux dans la vie quotidienne des gens. Une statistique montre par ailleurs que 20 % des internautes ont dépensé de l'argent réel dans un jeu, majoritairement des femmes (selon PopCap Games).

## Quelques géants du Social Gaming

### Facebook

Bien sûr, le succès de tous les acteurs du Social Gaming suscite la convoitise, principalement celle du réseau social Facebook lui-même. En juillet 2011, une monnaie virtuelle est imposée. Le principe en est simple : le joueur achète des « crédits Facebook » avec de l'argent réel, utilisables dans le jeu après l'achat. Le réseau social y trouve un intérêt majeur puisqu'il perçoit une commission de 30 % sur chaque transaction effectuée, augmentant son chiffre d'affaires de plusieurs centaines de millions d'euros.

### Zynga

Avec 600 millions de dollars de chiffre d'affaires, Zynga décroche sans aucun complexe la première place dans les entreprises présentes sur le Social Gaming. Son succès lui a permis de s'introduire en bourse en décembre 2011 à 1 milliard de dollars. Depuis, en moins d'un an, Zynga a perdu beaucoup de sa valeur, et a dû se résoudre à lancer un plan social. Ses jeux les plus connus sont CityVille, FarmVille, PlayFish ou Mafia Wars et totalisent 350 millions de joueurs sur Facebook en 2011, dont 80 millions uniquement pour le jeu FarmVille.

Pour satisfaire tous ces joueurs et s'adapter à toutes les plates-formes, Zynga a racheté l'entreprise Dextrose en 2011, spécialisée dans le HTML5.

### Disney

Avec Rocketpack, Disney s'affranchit de la technologie Flash d'Adobe, qui est devenue l'un des standards de fait du jeu en ligne. De plus, au fur et à mesure de l'adoption du HTML5, le groupe peut aussi passer outre l'écosystème exigeant d'Apple pour la

distribution d'applications. Le groupe peut enfin imaginer des portages sur toutes les plates-formes, du PC aux terminaux mobiles, en passant par les tablettes tactiles.

#### TECHNOLOGIES RocketPack

Rocketpack est un moteur de jeu en HTML5 qui a fait naître, fin 2010 sur Facebook, le premier jeu dans cette technologie : Warimals. L'entreprise Rocketpack, basée à Helsinki en Finlande, a misé sur cette technologie pour un jeu disponible sur plusieurs plates-formes.

#### Electronic Arts

En novembre 2009, Electronic Arts a racheté Playfish pour 275 millions de dollars. Son principal jeu est Sims Social, reprenant le concept des jeux Sims sur PC, avec plus de 50 millions de joueurs. Concurrencé fortement par The Ville de Zinga, Electronic Arts veut prendre une place dans le milieu du Social Gaming. Son modèle commercial agressif incite les joueurs à dépenser malgré la gratuité des jeux. D'autres jeux très connus sont adaptés sur Facebook, notamment Fifa Superstars, un jeu de simulation de football où une bonne configuration de l'équipe offre une chance de gagner le match.

### Étude démographique et comportementale

D'après une étude réalisée en 2011 par PopCap Games, concepteur de jeux vidéo, les internautes aux USA et au Royaume-Uni jouent significativement plus aux jeux sociaux depuis janvier 2010 : 41 % des internautes ont joué à un jeu social au cours des 3 derniers mois et plus de 15 minutes par semaine, contre 24 % en janvier 2010.

**Tableau 3-1.** Proportion du temps consacré au jeu social en 2010 et 2011 aux États-Unis et au Royaume-Uni (selon PopCap Games)

	États-Unis	Royaume-Uni	États-Unis	Royaume-Uni
	2010	2011	2010	2011
<b>Au cours des 3 derniers mois</b>	28 %	42 %	29 %	42 %
<b>Au moins 15 minutes / semaine</b>	24 %	41 %	25 %	41 %
<b>Au moins 6 heures / semaine</b>	8 %	16 %	6 %	13 %

Contrairement à ce que l'on pourrait penser, les internautes jouant sur les réseaux sociaux sont surtout des femmes (69 %) et l'âge moyen est de 39 ans (2011).

**Tableau 3-2.** Proportion des joueurs en 2010 et 2011 par sexe aux États-Unis et au Royaume-Uni (selon PopCap Games)

	États-Unis	Royaume-Uni	États-Unis	Royaume-Uni
	2010	2011	2010	2011
<b>Homme</b>	46 %	46 %	42 %	42 %
<b>Femme</b>	54 %	54 %	58 %	58 %

**Tableau 3-3.** Proportion des joueurs en 2010 et 2011 par âge aux États-Unis et au Royaume-Uni (selon PopCap Games)

	États-Unis	Royaume-Uni	États-Unis	Royaume-Uni
	2010	2011	2010	2011
<b>Moyenne</b>	45,0 ans	41,2 ans	38,0 ans	35,5 ans
<b>21 et en dessous</b>	4 %	9 %	9 %	9 %
<b>22-29</b>	11 %	21 %	22 %	25 %
<b>30-39</b>	20 %	17 %	25 %	26 %
<b>40-49</b>	20 %	14 %	22 %	21 %
<b>50-59</b>	26 %	18 %	15 %	13 %
<b>60+</b>	20 %	20 %	8 %	7 %

Les raisons de jouer à ces jeux sont bien entendu avant tout de s'amuser, mais aussi d'entrer en compétition contre les amis et de se déstresser du travail quotidien.

**Tableau 3-4.** Proportion des joueurs en 2010 et 2011 par raison (selon PopCap Games)

	2010	2011
<b>Pour s'amuser</b>	53 %	57 %
<b>Par esprit de compétition</b>	43 %	43 %
<b>Pour se déstresser</b>	45 %	42 %
<b>Pour l'entraînement mental</b>	32 %	32 %
<b>Pour accomplir des objectifs</b>	20 %	24 %
<b>Communiquer avec d'autres personnes</b>	24 %	24 %
<b>Améliorer la coordination main-yeux</b>	10 %	9 %

<b>Exprimer ma personnalité</b>	7 %	7 %
<b>Interagir avec des contacts du réseau social</b>	5 %	6 %

Le nombre des joueurs qui achètent de la monnaie virtuelle avec de l'argent réel a nettement augmenté de janvier 2010 (14 %) à 2011 (26 %).

**Tableau 3-5.** Proportion des joueurs qui font un achat en 2010 et 2011 par sexe aux États-Unis et au Royaume-Uni (selon PopCap Games)

	États-Unis	Royaume-Uni	États-Unis	Royaume-Uni
	2010	2011	2010	2011
<b>Gagner ou dépenser la monnaie virtuelle</b>	55 %	62 %	48 %	53 %
<b>Achat de la monnaie virtuelle avec de l'argent réel (% du total)</b>	16 %	27 %	12 %	25 %
<b>Achat d'un cadeau virtuel pour quelqu'un</b>	35 %	31 %	28 %	34 %

## Etude technologique

L'ordinateur de bureau était beaucoup utilisé en 2010, mais il est de plus en plus fréquemment remplacé aujourd'hui par les tablettes tactiles. Un milliard de smartphones est prévu pour 2015-2016. Créer un jeu en HTML5 pour les différentes plates-formes revient à toucher un nombre d'utilisateurs non négligeable ; en France, 12,8 millions de personnes jouent sur les téléphones mobiles (selon afjv.com).

**Tableau 3-6.** Proportion des technologies utilisées en 2010 aux États-Unis et au Royaume-Uni (selon PopCap Games)

	États-Unis	Royaume-Uni
<b>Ordinateur</b>	96 %	92 %
<b>Téléphone standard ou compatible Web</b>	8 %	13 %
<b>Smartphone</b>	28 %	29 %
<b>Console de jeu</b>	20 %	19 %
<b>Tablette tactile</b>	12 %	8 %

## Votre étude de marché

Le monde du jeu en ligne est en perpétuelle évolution, aussi les noms et chiffres que nous venons de citer évolueront également, souvent très rapidement. Avant de vous lancer dans quelque développement que ce soit, il est important que vous établissiez votre propre étude. Connaître le milieu dans lequel vous allez vous lancer (ses acteurs, ses composantes technologiques et économiques) est de la première importance pour l'étape suivante : votre positionnement par rapport à vos concurrents.

## Positionnement

Le positionnement du jeu est important pour justifier son prix – dans le cas d'un jeu payant – mais surtout pour se démarquer de la concurrence.

Plusieurs questions doivent être posées :

- Comment me démarquer de la concurrence selon l'étude de marché?
- À qui s'adresse le jeu? Quelles sont les attentes?
- Quel est le prix du jeu?

Il ne serait pas judicieux de reprendre un concept existant par manque d'information. Il ne serait pas non plus envisageable de créer un jeu avec une difficulté intellectuelle trop élevée s'il s'adresse à des enfants de moins de huit ans.

## Joueurs ciblés

Lorsque vous ciblez des joueurs, il faut connaître plusieurs points :

- leur age;
- leur sexe;
- leurs centres d'intérêt;
- leur temps disponible;
- leur technologie préférée.

L'âge influe non seulement sur la difficulté du jeu, mais aussi sur le type de graphismes employé. Par exemple, un jeu très coloré sera plus approprié pour des joueurs jeunes, voire très jeunes (moins de 8 ans); il pourra s'orienter vers le jeu d'apprentissage, type très apprécié par les parents (n'oublions pas que ce sont eux qui achèteront... ou non).

**REMARQUE Facebook interdit aux enfants de moins de 13 ans aux États-Unis**

Officiellement, aux États-Unis, le texte COPPA (Children's Online Privacy and Protection Act) protège les enfants de moins de 13 ans de la récolte de leurs données privées à des fins commerciales. Ainsi, l'enfant doit demander l'autorisation à ses parents, procédure longue et fastidieuse. Facebook a donc choisi de refuser l'accès à ses services aux enfants de moins de 13 ans. Tenez-en compte dans votre positionnement!

**Figure 3-1**

Univers coloré et enfantin  
du jeu ClickySticky sur iPad



Le sexe a une incidence sur l'univers du jeu. Sans tomber dans les clichés où un jeu ayant pour but de chercher des chaussures à talon aiguille dans un univers rose bonbon est destiné à un public féminin, un jeu conforme à leurs centres d'intérêt a des chances d'attirer plus de joueuses. Croire qu'il y a plus d'hommes jouant à des jeux sociaux est une pensée infondée; d'après une étude de MocoSpace, les joueuses y sont plus présentes (comme The Sims Social) que sur les consoles de salon.

**Figure 3-2**

The Sims Social



## Prix du jeu

Pour déterminer un prix, il faut prendre en compte plusieurs critères :

- le prix de la concurrence ;
- le prix attendu par les joueurs potentiels ;
- le rapport qualité / prix du jeu.

Paradoxalement, un prix faible nuit parfois à l'image d'un jeu car les joueurs potentiels peuvent sous-estimer la qualité. Estimez un prix convenable, du même ordre de grandeur que ceux de la concurrence, tout en pensant à la rentabilisation de votre travail ou de votre équipe.

À l'inverse, des jeux gratuits attirent plus facilement les joueurs. Une solution souvent adoptée consiste à proposer quelques niveaux gratuitement et la suite du jeu payante, ou encore le jeu gratuit seulement pendant une période d'essai au-delà de laquelle il faudra payer pour continuer à s'en servir. Pour un jeu de Social Gaming, pensez à la monnaie virtuelle ; le jeu est gratuit mais des extensions, des objets ou bonus deviennent payants.

### Renvoi Monnaie virtuelle

Reportez-vous au dernier chapitre pour l'intégration de la monnaie virtuelle dans votre jeu.

## Type du jeu

Le concept doit également tenir compte du type du jeu. Il faut se souvenir que certains types de jeux sont plus longs à produire, plus compliqués à réaliser que d'autres : par exemple, le scenario d'un jeu de rôle implique de passer beaucoup de temps à créer un univers riche en détails, alors qu'un jeu de simulation ou de gestion ne demande que la création d'algorithmes sans pour autant rentrer dans des développements de moult scènes complexes.

Voici quelques conseils :

- Choisissez le type de jeu selon vos moyens financiers.
- Estimez correctement le temps et les capacités qui vous seront nécessaires ; ne vous lancez pas dans un projet qui risque de vous dépasser et que vous ne terminerez peut-être pas.
- Si le scénario est la racine de votre jeu, réfléchissez sur son attrait auprès des joueurs avant de vous lancer dans le développement.
- Dans votre étude de marché, prenez en compte le type de jeu le moins présent. Vous pourrez vous démarquer de la concurrence.
- Sachez qu'un concept n'a pas besoin d'être complexe pour devenir très populaire.
- Dans le cadre du développement, informez-vous sur les outils disponibles pour créer le type de jeu retenu.

- Si vous souhaitez créer un jeu multijoueur, tenez compte la masse de travail. Par exemple, créer un jeu de simulation ne demande pas autant de travail qu'un jeu de rôle MMORPG (voir ci-après), dans lequel il faut intégrer des univers, des personnalisations de personnages, des modes de combat...;
- Nous présentons succinctement les divers types de jeux. N'hésitez pas à chercher plus d'informations dans des livres spécialisés ou sur Internet. Cela peut vous donner des idées pour votre concept.

## Action

Le jeu d'action demande au joueur dextérité et réflexes pour avancer dans les différents niveaux. Il en existe plusieurs sortes :

- FPS (*First Person Shooter*) : jeu avec une vue subjective dans un milieu 3D (ex. Doom);
- Combat : par des combinaisons de touches, le joueur doit combattre l'intelligence artificielle (ex. Street fighter);
- *Beat Them All* : jeu caractérisé par un nombre important d'ennemis dans le niveau, que le joueur devra tous éliminer. Pour augmenter la difficulté, les ennemis interviennent successivement pour combattre le héros (ex. Devil May Cry).

## Jeu de rôle

Dans ce type de jeu, aussi appelé RPG (*Role Playing Game*), le joueur doit faire évoluer un personnage dans un univers pour améliorer ses caractéristiques (sa force, sa défense, son attaque...) et apprendre de nouvelles techniques et compétences lorsqu'un niveau est gagné. Ces dernières sont obtenues par l'intermédiaire de points d'expérience, acquis principalement en combattant des ennemis plus en plus difficiles selon la progression du joueur.

Le héros du jeu progresse dans l'histoire en parlant au personnage non joueur (PNJ) qui lui indique des chemins, donne des objets indispensables ou déclenche de nouvelles actions possibles dans le jeu.

Généralement, le héros possède une somme d'argent, gagnée lors des combats ou trouvée dans des endroits spécifiques sur la carte, qui sert à acheter des armes plus puissantes et des montures plus résistantes pour vaincre des ennemis difficiles.

Les RPG sont regroupés en deux catégories :

- Japanese RPG : jeu basé principalement sur le scénario, souvent linéaire à cause de l'histoire fixée dès le début; personnages possédant une personnalité propre (ex. Final Fantasy);
- Occidental RPG : univers plus ouvert que dans les J-RPG, laissant au joueur une plus grande liberté, aussi bien sur le choix des personnages que sur le scénario (ex. série des The Elder Scrolls).

Traditionnellement, un RPG propose un combat par tour. Le joueur peut prendre le temps de choisir ses actions pour attaquer l'ennemi. Lorsque son tour est terminé, c'est à l'adversaire d'effectuer la même démarche. Cependant, il existe d'autres modes de combat :

- A-RPG (*Action-RPG*). Le combat s'effectue en temps réel. Le personnage doit attaquer l'ennemi sans prendre le temps de choisir une technique précise. C'est principalement des combinaisons de touches qui lui permettront d'attaquer (ex. Kingdom Hearts).
- T-RPG (*Tactical-RPG*). Chaque personnage est caractérisé par une zone de déplacement et une zone d'attaque. Comme dans le jeu des échecs, la stratégie entre en vigueur. Le terrain et la notion de déplacement sont des paramètres supplémentaires dans le déroulement du combat (ex. Final Fantasy Tactics).

Dans le cadre du multijoueur, on appelle ce type de jeu MMORPG (*Massive Multi-player Online Role Playing Game*). Le principe est le même, mais c'est l'aspect interactif entre les joueurs qui est mis en avant. Par exemple, des groupes peuvent se former pour appliquer des tactiques communes, mais aussi pour échanger, acheter ou vendre des objets entre partenaires (ex. Dofus).

## Aventure

Le jeu d'aventure classique est le Point&Clic. Ce mode est basé sur l'exploration d'un univers sans exiger aucune habileté du joueur ; c'est sa réflexion qui est sollicitée pour comprendre des situations, trouver des objets, créer des combinaisons afin de poursuivre son aventure.

Ce type de jeu est très intéressant pour les nouvelles plates-formes. En effet, les tablettes tactiles s'acquittent bien de l'interaction entre un joueur et un décor : l'ergonomie se prête bien à cette utilisation, donnant une expérience de jeu plus agréable (ex. Les Chevaliers de Baphomet).

## Action-Aventure

Comme son nom l'indique, ce type de jeu conjugue l'action et l'aventure, c'est-à-dire la résolution de problèmes dans un univers ouvert pouvant être exploré par le joueur. Les actions se déroulent en temps réel. Le mode de combat A-RPG s'inspire de ce genre (ex. The Legend of Zelda).

## Simulation

Le jeu de simulation est assez utilisé dans le Social Gaming. Le joueur a la capacité de s'adapter dans un environnement pré-défini, de réfléchir à la gestion de son avancée pour atteindre un objectif. Beaucoup de jeux de simulation intègrent une monnaie virtuelle que le joueur doit bien gérer pour progresser dans le scénario (ex. The Sims City).

## Sport

Un sport particulier est représenté dans le jeu. Le joueur doit diriger une équipe ou un personnage contre un adversaire en suivant les règles de ce sport (ex. Fifa).

## Réflexion

Les jeux de réflexion ne se basent pas sur la dextérité du joueur, contrairement aux jeux d'action, mais sur leurs capacités intellectuelles pour atteindre un but ou établir une stratégie ; l'aléatoire est très peu présent. Il s'agit principalement de jeux de société (échecs, dames...) et de quelques jeux de simulation.

## Choisir un framework

Vous n'êtes pas le premier à souhaiter concevoir et réaliser un jeu en ligne. D'autres avant vous se sont frottés à un certain nombre de difficultés récurrentes (graphismes, animations, contrôles...) et ont développé des outils (frameworks) facilitant considérablement le travail.

Plusieurs frameworks existent pour la réalisation d'un jeu en HTML5 ; certains privilient le dessin, d'autres les animations, ainsi que les contrôles. Aucun framework ne sort réellement du lot. Mettez-les à l'épreuve et ciblez celui vraiment approprié à vos besoins.

### FRAMEWORK Comment choisir ?

Dans la majorité des cas (pour ne pas dire tous), les frameworks ont été conçus par des développeurs indépendants. Il n'y a pas vraiment de framework en premier plan. Easel.js (voir annexe A) est intéressant mais son évolution est assez lente (de la version 0.4.2 vers la version 0.5 après plus de 6 mois). Kinetic commence à avoir un soutien mais il est très orienté vers le dessin sur HTML5 [Canvas](#) et moins sur le jeu vidéo (Scrolling, Level design, contrôle, etc.).

Il est important de choisir un outil qui corresponde à la fois au type de jeu que vous souhaitez développer et à vos goûts en matière de programmation.

## Easel.js

Easel.js est une bibliothèque assez poussée où la syntaxe de l'API est proche de Flash. Les éléments sont placés hiérarchiquement et intègrent une interactivité avec la souris. Des notions supplémentaires améliorent le framework :

- les animations ;
- les SpriteSheets (fichiers images regroupant plusieurs des illustrations d'un site, que l'on « découpe » ensuite pour les utiliser) ;
- les filtres : application d'effets de couleur et de floutage sur un élément.

► <http://www.createjs.com>

Par ailleurs, Easel.js s'accompagne d'autres bibliothèques :

- Tween.js : animation de déformation durant un temps (basé sur une frise temporelle virtuelle);
- Sound.js : gestion des sons et musiques;
- Preload.js : gestion des pré-chargements avant le commencement du jeu.

#### RENOVI Pré-chargement

Nous parlerons de l'intérêt du pré-chargement au cours du chapitre 2.

#### RENOVI Apprendre à utiliser Easel.js

L'annexe B explique comment se servir de Easel.js.

## Kinetic.js

Ce framework utilise plusieurs couches définies par l'utilisateur, chacune possédant deux contextes :

- l'affichage;
- une couche pour la détection des événements (clic, passage de la souris).

La scène est composée de nœuds virtuels, semblables au DOM, intégrant des formes et des images.

#### RESSEMBLANCES Flash est toujours là...

Les nœuds virtuels de Kinetic.js ressemblent beaucoup à Flash, ses scènes et ses MovieClip. En fait, tous les frameworks reprennent la structure de Flash. La différence avec Easel.js est la syntaxe : Easel.js reprend les noms des classes de Flash.

Kinetic.js est une surcouche de `canvas` qui permet de gagner du temps dans le dessin de formes complexes :

- Cercles;
- Polygones;
- Ellipse;
- Étoile.

La notion des Sprites ainsi que les animations sont présents dans le framework et utiles pour la création des jeux.

► <http://www.kineticjs.com>

## Crafty.js

Framework traditionnel pour créer des jeux en HTML5, Crafty.js incorpore une gestion des scènes (pause, réglage, etc.) et d'autres fonctionnalités :

- SpriteSheet;
- gestion des collisions ;
- gestion du clavier ;
- gestion d'une frise temporelle virtuelle ;
- animations ;
- gestion de la sauvegarde avec plusieurs solutions : IndexedDB, WebSQL, LocalStorage ou Cookies ;
- gestion du pré-chargement.

Détail supplémentaire, Crafty.js vous laisse le choix entre la création avec HTML5 `Canvas` ou en DOM/CSS (en gardant le même code).

► <http://craftyjs.com>

## Caat.js

Caat propose d'utiliser la technologie de votre choix : HTML5 `Canvas`, DOM/CSS ou WebGL. Dans tous les cas, l'API restera identique :

- animations ;
- gestion du clavier ;
- création de chemin : l'élément suit le chemin avec une vitesse prédéfinie (interpolation de mouvement) ;
- SpriteSheet ;
- gestion du pré-chargement ;
- filtres ;
- interpolation et transformation.

► <http://labs.hyperandroid.com/static/caat/>

## CanvasEngine.js

Utilisé dans cet ouvrage, CanvasEngine est un jeune framework orienté vers les jeux en HML5. Sa particularité est de se servir d'une structure prédéfinie pour les scènes du jeu. Elle intègre le préchargement et l'affichage en boucle des éléments toutes les 60 images par seconde (avec `requestAnimatonFrame`).

La hiérarchie des éléments est semblable au DOM. Enfin, un modèle pour la conception d'un jeu multijoueur est présent. D'autres fonctionnalités permettent l'élaboration du jeu :

- animations;
- frise temporelle virtuelle;
- gestion du clavier et de la manette;
- gestion des sons;
- gestion du défilement des cartes (*Scrolling*);
- intégration des cartes créées avec Tiled Map Editor;
- gestion de la sauvegarde;
- SpriteSheet.

► <http://canvasengine.net>

## Installer CanvasEngine

Dans la suite de l'ouvrage, nous utiliserons CanvasEngine pour la création d'un jeu vidéo en HTML5. L'idée est d'avoir un framework facilitant le travail en intégrant des notions basiques comme le défilement ou la création des niveaux. Qui plus est, le framework définit une structure bien précise pour la conception d'un jeu et vous prépare à l'étendre vers le multijoueur en séparant le calcul – le modèle – l'affichage – la vue.

Pour l'installer, suivez ces étapes :

- 1 Téléchargez la dernière version du framework.
- 2 En en-tête de la page, insérez le code suivant :

### Insertion de CanvasEngine

```
<script src="canvasengine-X.Y.Z.all.min.js"></script>
```

X, Y et Z étant les numéros de la dernière version. Le prochain chapitre explique comment initialiser CanvasEngine.

# 2

## Créer l'écran titre

---

Avant de commencer le jeu, le joueur appréciera le dynamisme des écrans titres. Comment charger les graphismes et diriger le joueur vers le jeu ?

L'écran titre est le premier visuel de votre jeu. Le joueur aura un avant-goût de la qualité et des possibilités de votre produit. Ne négligez donc pas cette partie. Le chargement au départ évitera les attentes ultérieures dans le jeu ; pensez donc à charger les images récurrentes en vérifiant leur bonne résolution.

## Qualité des images

Une réflexion préalable est nécessaire sur la qualité des images du jeu. Plus leur résolution est haute, plus le joueur appréciera la beauté du jeu. Mais qui dit forte résolution, dit fichier volumineux ! Comment proposer la meilleure qualité sans que le temps de chargement en pâtitse ?

Cette réflexion est d'autant plus importante que le chargement sur le Web n'est pas natif. Chaque ressource étant téléchargée sur un serveur, imaginez le temps d'attente si les images font plusieurs mégaoctets ! Par ailleurs, les utilisateurs sur mobiles vous sauront gré de ne pas consommer tout leur forfait avant d'avoir commencé à jouer.

L'idée est de proposer une résolution selon la nature de la plate-forme utilisée par le joueur. Sur un mobile ayant une résolution de  $854 \times 480$  pixels, il n'est pas forcément nécessaire d'afficher des images HD au détriment du chargement. A contrario, sur une tablette tactile avec un écran Retina (comme le Nouvel iPad), les images HD rendent une expérience de jeu plus appréciable.

Ainsi, nous aurons deux dossiers :

- les images HD ;
- les images SD.

Une problématique apparaît dans cette façon de procéder. Si nous augmentons l'échelle de l'affichage, cela signifie que les positions des éléments sur l'écran ne sont plus les mêmes. Dans un premier temps, on aurait le réflexe de redimensionner avec des méthodes de HTML5 telles que `scale`. Toutefois, il ne faut pas oublier que les images sont déjà redimensionnées ! Il faudrait donc appliquer un calcul de ratio sur les positions de chaque élément pour avoir un affichage correct sur les différentes résolutions. Nous verrons par la suite qu'il s'agit là d'une fausse bonne idée.

**Figure 2-1**

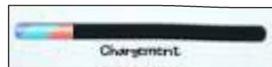
Sur le jeu Cute The Rope (en HTML5), il est proposé au joueur de choisir sa résolution en bas à gauche sur l'écran titre.



## Chargement initial

Pour un meilleur confort d'utilisation, il est judicieux de charger toutes les images dès le lancement du jeu. Pour optimiser le chargement, nous utilisons des images composées de plusieurs Sprites nommées SpriteSheets (voir l'aparté plus loin)

**Figure 2-2**  
Barre de chargement classique



## Création de la scène de chargement

Dans un premier temps, créons une scène comportant quelques images, comme une barre de chargement.

### La scène du chargement

```
canvas.Scene.new({
    name: "Preload", ①
    materials: { ②
        images: {
            background_preload ③: "images/sd/preload/background.png",
            bar_empty: "images/sd/preload/empty.png",
            bar_full: "images/sd/preload/full.png"
        }
    },
    ready: function(stage) { ④
    }
});
```

Le nom de la scène ① va nous servir pour son appel. Nous indiquons la liste des images ② à charger en précisant un identifiant à chaque fois ③. La méthode `ready` est appellée dès que la scène est prête; autrement dit, lorsque les images sont chargées ④. Le paramètre `stage` est l'élément principal initialement lié à la scène. Nous pouvons ensuite lui ajouter des éléments enfants avec la méthode `append`. Nous détaillerons ultérieurement les ajouts des images.

## Schéma des données

- 1 Pour séparer le code et les données, tous les chemins vers les images sont rentrés dans un fichier JSON qu'on importera avec une requête Ajax.
- 2 Le fichier JSON est stocké dans un dossier `Data` et est composé du schéma suivant :

### Contenu du fichier JSON

```
[
    {
        "id": "_background",
        "path": "level_1/LEVEL_01_DECOR_PLAN_03.png"
    },
    {
        "id": "foreground",
        "path": "level_1/LEVEL_01_DECOR_PLAN_01.png"
```

```

    },
    ...
]
```

L'intérêt est de charger des images d'autres scènes dans l'écran de pré-chargement. Notre fichier JSON sera appelé dans la méthode `ready` de la scène courante.

## Affichage de la barre de progression

3 Nous pouvons commencer à charger du jeu dans cette scène. À chaque fois qu'une image est chargée, nous rafraîchissons la scène pour afficher l'avancement de la barre de progression.

**Figure 2-3**  
Barre de chargement avec une image de fond (Angry Birds)



## Image de fond

4 Avant tout, pour la beauté du jeu, nous affichons une image de fond fixe.

### Ajout de l'image de fond

```

var self = this,
    el = this.createElement(); ①
el.drawImage("background", 0, 0); ②
stage.append(el); ③

```

Créer un élément se fait avec la méthode `createElement` ①. Ensuite, nous le récupérons pour effectuer des manipulations avec les même méthodes que HTML5 Canvas (simple surcouche).

Remarquez que nous assignons l'identifiant de l'image chargée de la scène dans la méthode `drawImage` ②.

Pour terminer, nous ajoutons l'élément dans la scène actuelle avec la méthode `append` ③. L'élément devient un enfant de la scène. Ainsi, si nous modifions la scène, les enfants prendront les effets de cette modification. Le principe est donc équivalent à celui du DOM d'une page. Le rafraîchissement se fait dans la méthode `render` :

### Rafraîchissement dans la scène

```
render: function(stage) {
    stage.refresh();
}
```

La méthode `render` est appelée à chaque rafraîchissement. Nous mettons à jour les éléments de la scène en appliquant la méthode `refresh` sur l'élément principal.

### Barre de progression

#### Affichage et animation de la barre de progression

```
CE.getJSON("data/materials.json", function(files) { ①
    var percentage = 0,
        bar_full = self.createElement(), ②
        bar_empty = self.createElement(); ③
    bar_empty.drawImage("bar_empty", 215, 250);

    stage.append(bar_empty);
    stage.append(bar_full);

    canvas.Materials.load("images" ④, files ⑤, function() { ⑥
        percentage += Math.round(100 / files.length);
        bar_full.drawImage("bar_full", 215, 250, percentage + "%");
        stage.refresh();
    }, function() { ⑦
        canvas.Scene.call("Title");
    });
}, 'json');
```

Nous effectuons la requête Ajax, sur le fichier JSON dont nous avons parlé plus haut, pour récupérer la liste des images *externes* à la scène actuelle ①. Une variable `percentage` comptabilise le taux de progression des images.

Comme pour l'image de fond, nous créons deux éléments : un pour afficher l'image de la barre vide ② et un autre pour afficher l'image de la barre remplie ③.

Notez que nous utilisons la variable `self` initialisée précédemment et qui fait référence à la scène courante.

La méthode `load` de la classe `Materials` indique :

- le type de fichiers à charger ④ ;
- la liste des images (sous forme de couples identifiant/chemin dans le fichier JSON) ⑤ ;
- une fonction appelée quand le fichier est chargé ⑥ ;
- une fonction appelée quand la totalité est chargée ⑦.

Lorsque chaque fichier est chargé ⑥, on calcule le nouveau pourcentage pour l'injecter dans la méthode `drawImage`. Le quatrième paramètre de cette dernière – exclusif au framework – définit la largeur de la barre selon un pourcentage.

Nous terminons par le rafraîchissement visuel de la barre avec la méthode `refresh`. Quand tous les fichiers sont chargés, nous chargeons et affichons 7 la scène nommée `Title` (écran titre) créée et expliquée dans la section suivante.

### SANS FRAMEWORK Charger des images

Une classe native de JavaScript nommée `Image` permet de charger des images, l'une après l'autre. Nous allons l'utiliser dans une classe statique `Materials` qui comporte deux fonctions de rappel :

- `onLoad`, appelée quand le chargement de l'image est terminé;
- `onFinish`, exécutée lorsque toutes les images ont été chargées.

Un tableau de la forme suivante est envoyé à la classe statique `Materials` :

```
var materials = [
  {
    id: "background"
    path: "path/to/image.png"
  }
];
```

L'identifiant permet de récupérer l'image plus tard. La classe `Materials` se définit de la façon suivante :

```
var Materials = {};
Materials.images = {};
Materials.loadMaterials = function(assets, onLoad, onFinish) {
  var i=0;
  load();
  function load() {
    var img;
    if (assets[i]) {
      img = new Image();
      img.onload = function() {
        Materials.images[assets[i].id] = this;
        i++;
        if(onLoad) onLoad.call(this);
        load();
      };
      img.src = assets[i].path;
    }
    else {
      if (onFinish) onFinish.call();
    }
  }
}
```

Il faut déterminer le moment où toutes les images sont chargées pour lancer le jeu. Pour cela, on réalise une sorte de « récursivité asynchrone » sur la fonction privée `load`, que nous appelons continuellement jusqu'à la fin du tableau. La méthode `call` exécute la fonction passée en paramètre.

Nous stockons l'instance `img` dans la propriété `images` de la classe. Ainsi, pour afficher l'image dans le `canvas` :

```
var canvas = document.getElementById('canvas_id'),
    ctx = this.element.getContext('2d');
ctx.drawImage(Materials.images["background"], 0, 0);
```

Vous constatez que la procédure n'est pas évidente ; c'est pourquoi l'une des missions des frameworks est de simplifier cette tâche (PreloadJS, CanvasEngine).

## Écran titre

L'écran titre est la première interface interactive. En soi, le principe en est très simple : nous disposons d'un fond (fixe ou animé) et de boutons (pour commencer le jeu, charger une partie, expliquer les règles...).

Bien entendu, il n'y a pas d'écran titre type. Selon le jeu, il sera demandé au joueur de charger une partie (jeux de rôle) ou pas (jeux d'arcade). Cependant, dans tous les cas nous aurons des boutons déclenchant des actions.

**Figure 2-4**  
Écran titre classique  
(Cut The Rope)



### RAPPEL Pensons au tactile pour l'interaction

Rappelons que le jeu sera jouable sur les smartphones et tablettes. Les interactions étant tactiles, il faut penser à créer des boutons suffisamment larges pour les doigts.

## Création de la scène de l'écran titre

### La scène de l'écran titre

```
canvas.Scene.new({
  name: "Title",
  ready: function(stage) {
```

```

    },
    render: function(stage) {
        stage.refresh();
    }
});
```

Il n'y a aucune surprise sur la création de la scène. Nous ne chargeons aucune image puisque cette opération a déjà été réalisée dans la scène du chargement. La méthode `refresh` rafraîchit la scène pour changer la couleur des boutons en cas d'interaction.

#### SANS FRAMEWORK Rafraîchissement d'un rendu

Pour réguler les animations et l'affichage sur le navigateur, une méthode nommée `requestAnimationFrame` appelle une fonction toutes les 60 frames :

```

(function() {
    var requestAnimationFrame = window.requestAnimationFrame "" window.
    ➔ mozRequestAnimationFrame ""
                    window.webkitRequestAnimationFrame "" window.
    ➔ msRequestAnimationFrame; ①
})();

function step(timestamp) {
    console.log(timestamp);
    requestAnimationFrame(step);
}
requestAnimationFrame(step);
```

Selon les navigateurs, la méthode porte différents noms. Nous sélectionnons la bonne méthode avant de l'utiliser ①. La fonction `step` est appelé en boucle pour le rafraîchissement.

## Initialisation des boutons

Dans la méthode `ready`, initialisation de l'objet contenant les données de chaque bouton

```

var self = this,
    buttons = {
        play ①: {
            height: 128, ②
            click: function() { ③
                canvas.Scene.call("Jeu");
            }
        },
        bonus:{ 
            height: 62,
            click: function() {
                canvas.Scene.call("Bonus");
            }
        },
    },
```

```

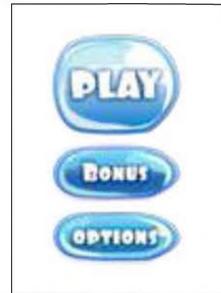
options: {
    height: 62,
    click: function() {
        canvas.Scene.call("Options");
    }
};

```

Chaque clé de l'objet ❶ contient les caractéristiques du bouton : sa hauteur ❷ (importante pour le découpage dans l'image comportant tous les boutons, voir figure), ainsi que sa fonction associée ❸.

**Figure 2-5**

Les boutons de l'écran sont, initialement, dans une seule image



Tous les boutons sont initialement collés dans la même image, que nous découpons selon les indications de hauteurs; chaque partie découpée (i.e. chaque bouton) se voit assigner les événements `click`, `mouseover` et `mouseout` et est affichée sur la scène.

Nous disposons d'une deuxième image du même type; c'est la couleur qui change, indiquant le type d'interaction avec le joueur.

#### Affichage des boutons

```

var pos = 0; ❶
for (var key in buttons) { ❷
    displayButton(buttons[key], pos);
    pos += buttons[key].height;
}

function displayButton(data_btn, pos) { ❸
    var width = 200,
        btn = self.createElement();
    btn.drawImage("buttons", 0, pos, width, data_btn.height, 0, 0, width,
    ↵ data_btn.height);
    btn.x = 50;
    btn.y = pos + 180;

    btn.on("click", data_btn.click);
    btn.on("mouseover", function() {
        this.drawImage("buttons_hover", 0, pos, width, data_btn.height, 0, 0,

```

```

    ↵ width, data_btn.height);
    });
    btn.on("mouseout", function() {
        this.drawImage("buttons", 0, pos, width, data_btn.height, 0, 0,
    ↵ width, data_btn.height);
        });
    stage.append(btn);
}

```

Par l'intermédiaire d'une boucle ②, nous appelons une fonction locale ③ affichant chaque bouton : nous créons un élément et nous affichons l'image du premier bouton à la position indiquée. La variable `pos` ① permet d'afficher les boutons l'un en dessous de l'autre ; elle est incrémentée à la fin de chaque tour de boucle.

#### RENOVI Découper une image

Voir l'annexe A sur la méthode `drawImage` pour comprendre ses paramètres.

## Association des événements

Associer un événement à un élément se fait avec la méthode `on`.

**Tableau 2-1.** Événement de la souris

Événement	Explication
<code>click</code>	Quand le joueur clique sur le bouton, la fonction est celle de l'objet initialisé précédemment.
<code>mouseover</code>	Quand le joueur passe la souris sur le bouton, nous changeons l'apparence de ce dernier avec l'autre image.
<code>mouseout</code>	Quand le joueur ne passe plus la souris sur le bouton, nous remettons l'image d'origine.

**SANS FRAMEWORK Récupérer les positions de la souris sur le canvas**

Les événements de la souris sur un élément du `canvas` résultent en fait d'un calcul d'intersection entre la position de la souris sur le `canvas` et celle de l'élément :

```
var canvas = document.getElementById('canvas_id');

function getMousePosition(e) {
    var obj = canvas, mouseX, mouseY,
        top = 0,
        left = 0;
    while (obj && obj.tagName != 'BODY') {
        top += obj.offsetTop;
        left += obj.offsetLeft;
        obj = obj.offsetParent;
    }
    mouseX = e.clientX - left + window.pageXOffset;
    mouseY = e.clientY - top + window.pageYOffset;
    return {x: mouseX, y: mouseY};
}
canvas.addEventListener('click', getMousePosition, false);
```

Nous ajoutons un écouteur de type `click` sur le `canvas`, pour renvoyer les positions de la souris. Pour cela, nous recherchons l'emplacement absolu de l'élément `canvas` en boucle sur la page.

Les anciens navigateurs ne reconnaissent pas les instructions `window.pageXOffset` et `window.pageYOffset` utilisées ici. Néanmoins, comme ils ne comprennent pas non plus HTML5, ils ne seront pas concernés par votre jeu. Vous pouvez donc utiliser sans crainte le code précédent dans votre jeu.

## Écrans additionnels

L'écran titre propose le plus souvent des boutons menant à des écrans additionnels dans lesquels le joueur peut choisir des options avant d'entamer sa partie ou encore commencer à jouer directement à un niveau particulier.

### Options

Généralement, un jeu offre des options :

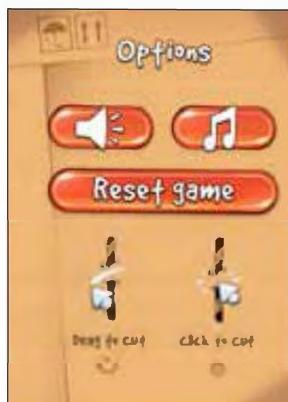
- réglage du volume sonore;
- affectation des touches du clavier et/ou de la manette;
- choix de la langue;
- niveau de difficulté;
- crédits.

Cette liste n'est pas exhaustive. Elle varie beaucoup selon votre type de jeu. Pour un jeu de simulation (comme la gestion d'une ville), vous pouvez ajouter la gestion du zoom, la vitesse du scrolling, du temps virtuel, etc.

Dans tous les cas, il faut penser à créer une scène pour cela et stocker les valeurs dans un cookie ou en local pour se souvenir des paramètres quand le joueur revient sur sa partie ultérieurement.

**Figure 2-6**

Exemple d'un écran d'options (Cute The Rope)



## Niveaux

Dans de nombreux jeux, il y a une gestion des niveaux : quand un niveau est terminé, le deuxième est débloqué et ainsi de suite.

Bien entendu, l'affichage des niveaux est propre au jeu. Cela peut s'apparenter à un monde à traverser ou une simple liste de niveaux.

Dans un jeu de rôle, le joueur arrive sur une carte de départ pour commencer à jouer sans passer par une scène de niveaux.

# 3

## Affichage des décors

---

Un jeu repose beaucoup sur les décors : disposition, aspect, animation, enchaînements, harmonisation de la structure des niveaux et cartes construisent une expérience de jeu agréable pour le joueur.

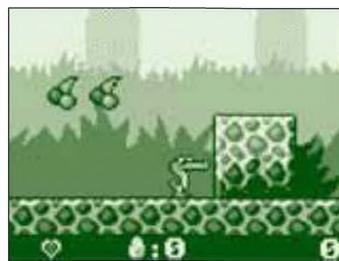
## Level Design

Le *Level Design* est le développement des différents niveaux selon le Gameplay (façon de jouer) et les règles du jeu établies. Techniquement, nous devons définir un schéma de données pour ensuite afficher les différents éléments sur l'écran.

Puisque nous souhaitons réaliser un jeu multijoueur par la suite, nous penserons dès maintenant à séparer d'un côté les données du jeu et de l'autre l'affichage.

**Figure 3-1**

Un niveau tiré du jeu  
Alex The Allegator



## Insertion des données

Créons donc le modèle comprenant les calculs.

### Création d'un simple modèle

```
Class.create("Game_Map", {
    loadMap: function() {
        // Chargement de la carte
    }
});
```

#### SANS FRAMEWORK Créer un modèle

Créer un modèle revient à créer une classe :

```
function Game_Map() {
    // Constructeur
}

Game_Map.prototype = {
    loadMap: function() {
        // Chargement de la carte
    }
};
```

Pour instancier une classe :

```
var game_map = new Game_Map();
```

Dans ce modèle, récupérons les données dans un fichier JSON et envoyons-les à la scène, qui les affiche.

#### REMARQUE Multijoueur

Pour le multijoueur, il est indispensable de séparer le calcul ou la récupération des données de l'affichage côté client. Le framework CanvasEngine prend en considération cette séparation même pour un jeu en local.

#### Scène chargeant la première carte

```
canvas.Scene.new({
    name: "map",
    ready: function(stage) {
        this.game_map = Class.new("Game_Map"); ①
        this.game_map.loadMap(); ②
    }
});
```

Nous instancions un attribut `game_map` ① de la classe créée précédemment. Un simple appel à la méthode `loadMap` ② permet de récupérer les données du modèle pour les afficher ensuite.

## Une carte

### Schéma

Dans un jeu de rôle, le décor est affiché isométriquement ou avec une vue de dessus. Dans les différents cas, la rotation de la carte change, mais ses données restent identiques.

Figure 3-2

Exemple d'une carte  
du jeu BrowserQuest



Nous allons créer un fichier JSON contenant le schéma de notre carte, c'est-à-dire sa structure.

Pour dessiner notre décor, nous nous servirons d'un tableau à trois dimensions :

- largeur de la carte;
- hauteur de la carte;
- couches de superposition.

Il nous faut également un *Tileset*, c'est-à-dire une image composée de tous les motifs à juxtaposer pour composer notre carte. Admettons que nous ayons une image de  $256 \times 192$  px composée de  $8 \times 6$  carreaux de base :

**Figure 3-3**

Image comprenant l'ensemble des carreaux



Ici, un carreau fait 32 px de côté. Sur le côté haut-gauche, nous avons plusieurs carreaux pour le sol de la carte. Sur la droite, d'autres carreaux, semblables à des rochers, viendront se superposer au sol. Le tableau sert à afficher notre carte en déterminant l'emplacement de chaque carreau.

**Figure 3-4**

La carte à afficher



Nous avons disposé un quadrillage pour bien visualiser les carreaux. Le code s'apparente à ceci :

#### Tableau pour définir la carte

```
[  
  [[0], [1], [2]],  
  [[8, 6], [9], [10]],  
  [[16], [17], [18, 6]]  
]
```

Les entrées du tableau correspondent chacune à un identifiant de carreau :

- 0 est le premier carreau de la première ligne.
- 8 est le premier carreau de la deuxième ligne.
- 16 est le premier carreau de la troisième ligne.
- 6 est le carreau du rocher de la première ligne.

Il est théoriquement possible d'« empiler » autant de couches qu'on le souhaite pour constituer notre carte ; nous nous limiterons à trois. Normalement, le tableau du schéma devrait donc avoir trois entrées. Si, pour une couche donnée, aucun motif ne vient remplir un carreau, on pourrait indiquer la valeur `null`. Cependant, pour ne pas alourdir le fichier JSON – surtout pour des cartes gigantesques ! - nous préférons ignorer les entrées pour les couches hautes. Les définitions dans notre tableau pourront donc prendre l'une des formes suivantes :

#### Empilement des motifs en un carreau donné

```
// Un carreau sur chacune des trois couches
[a, b, c]
// Un carreau seulement sur la deuxième couche
[null, b]
// Un carreau seulement sur la troisième couche
[null, null, c]
// Un carreau seulement sur la première couche
[a]
```

Chaque carreau doit posséder des propriétés (tableau de valeurs), comme son ordre de superposition ou sa praticabilité (gestion des collisions). Nous ajoutons ces données dans le schéma de la façon suivante :

#### Tableau pour définir la carte

```
{
  "map": [
    [[0], [1], [2]],
    [[8, 6], [9], [10]],
    [[16], [17], [18, 6]]
  ],
  "properties": {
    "0": [0],
    "1": [0],
    "2": [0],
    "6": [0] ❶,
    "8": [0],
    "9": [0],
    "10": [0],
    "16": [0],
    "17": [0],
    "18": [0]
  }
}
```

Par exemple, le carreau avec l'identifiant 6 ❶ (représentant le rocher) possède un tableau où la première valeur correspond à la superposition :

- 0 : en-dessous des personnages (type « sol ») ;
- 1 : au-dessus des personnages (personnage « caché » derrière le rocher).

Ici, le héros du jeu marchera sur le rocher. Nous utilisons un tableau car d'autres entrées seront ajoutées pour les collisions dans le chapitre 7.

#### Astuce Rendre le travail moins fastidieux

Comprendre le mécanisme des schémas pour construire un carreau ou une carte est bonne expérience dans la réalisation de jeux. Cependant, en production, ce travail peut devenir rapidement laborieux, sans compter le taux d'erreur humaine en rentrant de mauvais identifiants.

Pour vous aider, il existe des éditeurs de niveaux sur le Web. À titre d'exemple, nous vous présentons plus loin Tiled Map Editor, un éditeur OpenSource et gratuit pour créer des cartes pour des jeux RPG, de plate-forme ou de construction. À partir de cet éditeur, affichez votre carte selon le schéma proposé.

► <http://www.mapeditor.org>

Si aucun éditeur ne correspond à vos attentes, n'hésitez pas à en construire un vous-même ; il est cependant inutile de chercher la complexité, le minimum peut vous servir pour produire des niveaux rapidement.

### Affichage de la carte

Avant tout, créons deux constantes pour indiquer la taille en pixels d'un carreau et la largeur de l'image de référence (Tileset) : dans notre cas, respectivement 32 et 256 pixels.

#### Constante pour un carreau

```
var TILE_PX = 32;
var WIDTH_TILESET = 256;
```

Ensuite, nous chargeons le schéma dans le modèle.

#### Récupération du schéma JSON

```
Class.create("Game_Map", {
    loadMap: function(callback) {
        // Chargement de la carte
        CE.getJSON("map.json", callback);
    }
});
```

Puis, dans la méthode `ready` de notre scène, nous allons coder 3 boucles pour parcourir le schéma (une pour les couches, une pour parcourir les abscisses X et une dernière pour les ordonnées Y).

#### La scène de la carte

```
canvas.Scene.new({
    name: "map",
    materials: {
        images: {
            tileset: "tileset-fire.png"
        }
    },
},
```

```

ready: function(stage) {
    var self = this,
        map = this.createElement();

    // Création des couches
    this.layer = [];
    CE.each(7, function(i, val) {
        self.layer.push(self.createElement());
        map.append(self.layer[i]);
    });

    this.game_map = Class.new("Game_Map");
    this.game_map.loadMap(function(data) {

        var map_data = data.map,
            map_prop = data.properties;
        CE.each(map_data, function(i, x) {
            CE.each(map_data[i], function(j, array) {
                CE.each(map_data[i][j], function(k, id) {
                    if (map_data[i][j][k] === undefined) { ❶
                        return;
                    }
                    var tile = self.createElement();
                    var pos_y = id / (WIDTH_TILESET / TILE_PX) *
                    ↪ TILE_PX;
                    var pos_x = (id % (WIDTH_TILESET / TILE_PX)) *
                    ↪ TILE_PX;
                    tile.drawImage("tileset", pos_x, pos_y, TILE_PX,
                    ↪ TILE_PX, 0, 0, TILE_PX, TILE_PX); ❷
                    tile.x = i * TILE_PX; ❸
                    tile.y = j * TILE_PX; ❸
                    self.layer[k + (map_prop[id][0] * 3)].
                    ↪ append(tile); ❹
                });
            });
        stage.append(map);
    });
},
render: function(stage) {
    stage.refresh();
}
});

```

Il faut se souvenir que les éléments créés viennent se superposer sur ou sous les éléments précédents. Dans ce type de carte, des décors seront affichés en-dessous des personnages, d'autres au-dessus. Pour jouer sur ces superpositions, nous créons 7 couches :

- 3 couches de décor en-dessous des personnages;
- 1 couche de personnages;
- 3 couches de décor au-dessus.

#### MÉTHODE each

Comme jQuery, le framework possède une méthode nommée `each` pour effectuer une itération sur un tableau. Le premier paramètre de la fonction de rappel est l'index du tableau, le second sa valeur.

Toutes les couches sont stockées dans l'élément nommé `map` utilisé dans le défilement (scrolling). Voici les étapes de l'affichage :

- 1 Vérification de l'existence du carreau dans la couche par une simple condition `map_data[i][j][k] === undefined`.

#### RAPPEL Undefined

`undefined` est un type en Javascript indiquant qu'une variable n'est pas définie (aucune valeur affectée) :

```
var a;
console.log(a); // undefined
console.log(!a); // true
console.log(a == false); // false
console.log(!0); // true
```

Remarquez que la syntaxe de `!a` est attrayante pour vérifier si une valeur existe. Pourtant, si `a` vaut 0, la condition renverra le même résultat alors que 0 est pourtant une valeur fiable.

Dans le cas où nous avons besoin de valeur numérique, nous privilégions la condition `a === undefined` qui teste à la fois la valeur et le type.

- 2 Choix du carreau sur l'image de référence (tileset). L'identifiant est un nombre et non une position ; nous devons le transformer en une position (X,Y) sur l'image.
- 3 Positionnement du carreau. Il n'y a rien de complexe ; ce sont les valeurs des variables d'itération `i` et `j` multipliées par la taille du carreau qui définissent la position sur la carte.
- 4 Affectation du carreau à la bonne couche selon sa valeur de superposition. La variable `k` correspond à la couche dans le schéma. On lui ajoute une valeur supplémentaire, la superposition, pour indiquer si le carreau est en-dessous ou au-dessus des personnages.

## Un niveau

Pour un simple niveau, nous disposons de plusieurs éléments sur différents plans. Ces derniers défileront selon le mouvement du joueur (chapitre 6). Si tous les niveaux ont des ressemblances et si seul le *level design* change, il sera plus judicieux de programmer une scène générique.

## Le schéma

Nous souhaitons récupérer un fichier JSON écrit comme suit :

### Schéma de données pour un niveau

```
[  
  {  
    "img": "bottom",  
    "id": "bottom",  
    "polygon": [],  
    "layer": 2,  
    "data": [  
      {  
        "x": 4,  
        "y": 5,  
        "scale": 2,  
        "opacity": 0.9,  
        "rotation": 0  
      }  
    ]  
  },  
  ...  
]
```

Nous définissons tous les éléments présents dans le niveau avec les propriétés suivantes :

- `img` : l'identifiant de l'image préchargée
- `id` : l'identifiant de l'élément afin de spécifier son animation et son interaction plus tard;
- `polygon` : les points d'interaction de l'élément;
- `layer` : le plan où sera dessiné l'élément;
- `data` : l'élément est répété sur différentes positions avec des propriétés spécifiques comme la taille, l'opacité ou la rotation.

Le modèle se nomme `GameLevel` et la classe s'écrit de la même manière que le code du début de ce chapitre. En fait, lorsque le joueur choisit un niveau dans l'écran précédent, nous changeons la valeur du niveau courant. Ainsi, quand il arrive sur la scène du niveau, la requête Ajax récupéra les données du niveau choisi par le joueur.

### Le modèle pour les niveaux

```
Class.create("Game_Level", {  
  _levels: [ ①           {stars: 0, score: 0}  
  ],  
  _currentLevel: 0,  
  getCurrentLevel: function(data) {  
    if (data) {  
      return this._worlds[this._currentWorld][this._currentLevel]; ②  
    }  
  }  
})
```

```

        return this._currentLevel;
    },
    setCurrentLevel : function(id) {
        this._currentLevel = id;
    }
});

```

**À RETENIR** Le caractère « souligné » (**underscore**) dans les variables

En Javascript, pour différencier une variable privée, l'usage est de la préfixer par un caractère de soulignement (« \_ »).

Remarquez que la classe possède les caractéristiques de chaque niveau dans la propriété `_levels` ①. Ainsi, si la méthode `getCurrentLevel` ② reçoit un booléen à `true` dans le premier paramètre, elle renvoie les données du niveau et non son identifiant.

Nous pouvons désormais afficher les niveaux sur la scène.

#### La scène d'affichage

```

var WIDTH_BACKGROUND = 899; //899 : constante pour la largeur du fond.
canvas.Scene.new({

    name: "Level_Design",
    materials: {
        images: {
            background: "images/background.png"
            bottom: "images/bottom.png"
        }
    },
    ready: function(stage) {
        var self = this;

        this.game_level = Class.new("Game_Level");

        CE.each(3, function(i, val) {
            self.layer[val] = self.createElement();
            stage.append(self.layer[val]);
        });

        CE.each(2, function(i, val) {
            var background = self.createElement();
            background.drawImage("background");
            background.x = i * WIDTH_BACKGROUND;
            self.layer["1"].append(background);
        });

        CE.each(element, function(i, el) {
            CE.each(el.data, function(i, el_data) {
                var _el = self.createElement();

```

```

        _el.drawImage(el.img);
        _el.x = el_data.x;
        _el.y = el_data.y;
        self.layer[el.layer].append(_el);

    })
});

var level = this.game_level("getCurrentLevel");
CE.getJSON("data/level_" + (level+1) + ".json",
function(element) { ❶
    self.layer = {};

    CE.each(3, function(i, val) {
        self.layer[val] = self.createElement();
        stage.append(self.layer[val]);
    });

    CE.each(2, function(i, val) {
        var background = self.createElement();
        background.drawImage("background");
        background.x = i * WIDTH_BACKGROUND;
        self.layer["1"].append(background);
    });

    CE.each(element, function(i, el) {
        CE.each(el.data, function(i, el_data) {
            var _el = self.createElement();
            _el.drawImage(el.img);
            _el.x = el_data.x;
            _el.y = el_data.y;
            self.layer[el.layer].append(_el);

        })
    });
    stage.refresh();
});
},
render: function(stage) {
    stage.refresh();
},
}
);

```

CE.getJSON ❶ est une requête Ajax avec la méthode GET. Nous envoyons le niveau courant récupéré dans le modèle afin de récupérer le schéma d'un niveau bien précis.

Ensuite, nous effectuons trois étapes d'affichage :

- 1 création des plans;
- 2 mise en place de l'arrière-plan répétitif;
- 3 placement des éléments du schéma.

Nous créons trois plans stockés dans l'objet `layer`. Ce dernier est une propriété de la classe afin de l'utiliser dans les autres méthodes pour son déplacement.

## Utiliser Tiled Map Editor

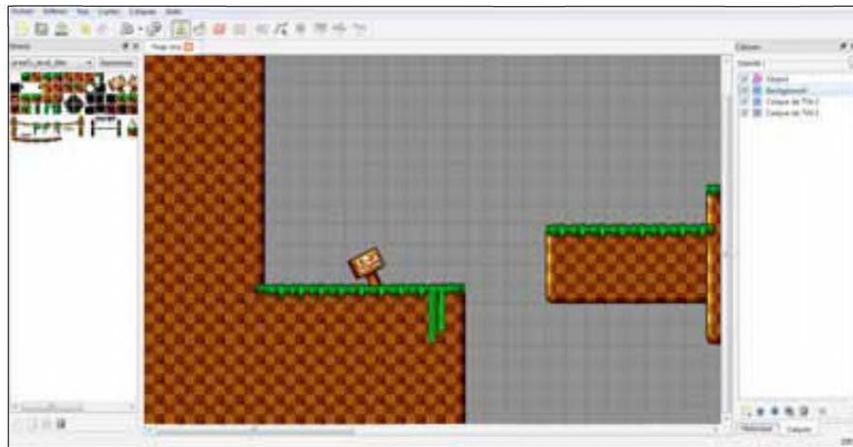
### Créer la carte

Construire un niveau ou une carte manuellement est long et fastidieux. Pour nous aider dans cette tâche, il existe un éditeur gratuit et OpenSource nommé Tiled Map Editor. Il aide à disposer automatiquement des carreaux à l'aide d'une image nommée `Tileset`, comme nous l'avons expliqué dans les sections précédentes.

#### TÉLÉCHARGEMENT Tiled Map Editor

► <http://www.mapeditor.org>

**Figure 3-5**  
Interface de Tiled  
Map Editor



Les calques définissent la superposition des décors. Plus le calque est en haut de la liste, plus l'élément qu'il représente est « au-dessus » des autres.

- 1 Cliquez sur l'icône *Nouveau*.
- 2 Choisissez la taille (en pixels) de la carte et celle d'un carreau, puis validez.
- 3 Dans la barre d'outils, cliquez sur *Carte et Nouveau Tileset*.

- 4 Le nom servira comme identifiant pour charger l'image dans CanvasEngine. Ajoutez l'image désirée.
- 5 L'image `Tileset` apparaît sur l'interface, découpée en plusieurs carreaux que vous pouvez sélectionner et placer sur la carte.
- 6 Pour créer une nouvelle couche, cliquez sur *Calques* et *Ajouter un calque de Tile*.
- 7 Enfin, pour avoir un calque contenant des personnages ou autres objets, cliquez sur *Calques* et *Ajouter un calque d'objet*.
- 8 Lorsque la carte est terminée, cliquez sur *Fichier* et *Exporter en tant que*. Enregistrez le fichier en format JSON dans un dossier de votre choix.

## Intégrer la carte dans la scène

Avant tout, ajoutons la classe `Tiled` dans l'initialisation du `canvas` :

### Ajout de Tiled dans l'initialisation

```
var canvas = CE.defines(["canvas"]).
    extend(Tiled).
    ready(function(ctx) {
});
```

Ensuite, déclarons le Tileset dans les ressources graphiques à précharger :

### Nom du Tileset

```
canvas.Scene.new({
  name: "map",
  materials: {
    images: {
      mytileset: "path/to/tileset.png"
    },
    ready : function() {
    }
});
```

Dans la méthode `ready` de la scène, nous allons insérer la carte et l'afficher directement dans un élément :

### Insertion de la carte de Tiled Map Editor

```
var tiled = canvas.Tiled.new();
tiled.load(this ①, stage ②, "map/map.json" ③);
tiled.ready(function() { ④
});
```

Nous chargeons la carte avec la méthode `load` en renseignant 3 paramètres :

- la scène ① (`this` fait référence à la scène courante) ;
- l'élément où sera dessiné le décor ② ;
- le chemin vers le fichier JSON ③.

La méthode `ready` ④ de la classe `Tiled` est exécutée dès que la carte est chargée et affichée.

## Obtenir des données de la carte

Plusieurs méthodes fournissent des informations à propos de la carte, par exemple pour connaître la taille des carreaux :

### Obtenir la taille des carreaux

```
tiled.ready(function() {
    var tile_w = this.getTileWidth(),
        tile_h = this.getTileHeight();
});
```

Si nous souhaitons insérer des éléments mobiles ou le personnage contrôlable par le joueur, nous devons insérer le calque d'objet :

### Insérer le calque d'objet

```
var player = this.createElement();
// ...

tiled.ready(function() {
    var layer_object = this.getLayerObject(); ①
        layer_object.append(player);
});
```

La méthode `getLayerObject` ① prend par défaut le calque d'objet le plus haut. Si vous en avez plusieurs, indiquez sa position en paramètre (à partir de 0).

## Objets principaux : les sprites

En mentionnant les objets principaux, nous parlons des personnages, des objets et tous les autres éléments qui ont une interaction avec le joueur ou une indépendance de leurs mouvements par rapport au déplacement de la carte.

Cette section mentionnera seulement comment insérer et afficher ces éléments ; l'interaction sera plus amplement développée dans le chapitre 7.

## Ensembles d'éléments graphiques ou Spritesheets

Un élément graphique de ce type est nommé *sprite*. Une image composée de plusieurs éléments graphiques pour effectuer des animations (selon une action du sprite) est appelé *Spritesheet*.

**Figure 3-6**

Plusieurs sprites pouvant former une animation



### QUESTION Pourquoi ne pas séparer toutes les images ?

Regrouper plusieurs éléments graphiques sur une seule image est important pour un jeu HTML5 qui, probablement, sera jouable en ligne. Le temps de chargement sera ainsi diminué. Ce n'est pas le poids des images qui rentre en jeu (séparées ou fusionnées, les images auront le même poids dans les deux cas), mais le temps d'attente des requêtes HTTP.

Une requête HTTP demande l'image au serveur, qui renvoie celle-ci au client. Cet aller-retour s'effectue une seule fois si les éléments graphiques sont sur une seule image. Généralement, le temps se compte en millisecondes et est donc imperceptible pour le joueur (contrairement au téléchargement). Cependant, un jeu peut avoir énormément d'images, surtout s'il est très riche graphiquement. Il n'est donc pas négligeable d'avoir des Spritesheets pour améliorer la performance du chargement.

CanvasEngine possède une classe `SpriteSheet` pour manier ces images aisément. Tout d'abord, insérez cette classe lors de l'initialisation du canvas :

#### Ajout de la classe `SpriteSheet`

```
var canvas = CE.defines("canvas").  
extend(SpriteSheet).  
ready(function(ctx) {  
  
});
```

Le `SpriteSheet` est préchargé et possède désormais un identifiant dans la création de la scène :

#### Chargement du Spritesheet

```
canvas.Scene.new({  
    name: "Scene",  
    materials: {  
        images: {  
            spritesheet: "images/spritesheet.png"  
        }  
    }  
});
```

```

        }
    },
    ready: function(stage) {
        }
    });
}

```

Voici notre image :

**Figure 3-7**

Plusieurs boutons pour réaliser un menu



La méthode ready va instancier une variable nommée `spritesheet` qui contiendra tous les éléments graphiques prêts à l'emploi.

#### Grille des différents éléments

```

this.spritesheet = canvas.SpriteSheet.new("buttons" ①, {
    grid: [
        size: [4, 5], ②
        tile: [68, 68], ③
        set: ["play_hover", "play", "zoom_p", "zoom_m"]
    ]
});

```

Le premier paramètre ① donne un identifiant à l'ensemble des sprites. Notre image est découpée en 4 lignes et 5 colonnes ② et un élément graphique mesure  $68 \times 68$  pixels ③. Ces données sont énumérées dans `size` et `tile`.

### SANS FRAMEWORK Insérer des sprites

Créons une classe `Spritesheet` semblable à celle du framework CanvasEngine :

```

function Spritesheet(image, set) {
    this.image = image;
    this._set = {};
    if (set) this.set(set); ①
}
Spritesheet.prototype = {
    set: function(set) {
        var gridset, gridname, x, y, grid_w, grid_h;
        for (var id in set) {
            if (id == "grid") {
                for (var i=0 ; i < set.grid.length ; i++) {
                    for (var j=0 ; j < set.grid[i].set.length ; j++) {
                        gridname = set.grid[i].set[j];
                        gridset = set.grid[i];

                        y = gridset.tile[1] * parseInt(j / Math.round(gridset.
                            size[1]-1));
                        x = gridset.tile[0] * (j % Math.round(gridset.
                            size[0]));

                        this._set[gridname] = [x ②, y ③, gridset.tile[0] ④,
                            gridset.tile[1] ⑤];
                    }
                }
            } else {
                this._set[id] = set[id];
            }
        },
        draw: function(ctx, id, dest) {
            dest = dest "" {};
            var tile = this._set[id];
            if (!tile) {
                throw "le Spritesheet " + id + " n'existe pas";
            }
            var dest_x = dest.x "" "0",
                dest_y = dest.y "" "0",
                dest_w = dest.w "" tile[2],
                dest_h = dest.h "" tile[3];

            ctx.drawImage(this.image, tile[0], tile[1], tile[2], tile[3], +dest_x,
                +dest_y, dest_w, dest_h); ⑥
        }
    }
}

```

Dans le constructeur, nous définissons la grille à découper et chaque carreau est sauvegardé dans la propriété `_set` ①. La valeur est un tableau contenant 4 informations :

- position X du sprite dans le Spritesheet ②;
- position Y du sprite dans le SpriteSheet ③;
- largeur du sprite ④;
- hauteur du sprite ⑤.

La méthode `draw` se sert de ces renseignements pour découper l'image et l'afficher dans le contexte ⑥. Pour utiliser la classe `Spritesheet` créée :

```
var canvas = document.getElementById("canvas_id"),
    ctx = canvas.getContext("2d");
var spritesheet = new Spritesheet(Materials.images["buttons"], {
  grid: [{{
    size: [4, 5],
    tile: [68, 68],
    set: ["play_hover", "play", "zoom_p", "zoom_m"]
  }]
});
spritesheet.draw(ctx, "play");
```

Remarquez que l'image est prise dans la classe `Materials` (voir chapitre 2).

#### Création d'un bouton à partir du Spritesheet

```
var btn = this.createElement();
btn.x = 70;
spritesheet.draw(btn, "play");
stage.append(btn);
```

Nous créons un élément en lui affectant l'image découpée, plus précisément le premier sprite en haut à gauche de l'image principale.

La méthode `draw` utilise `drawImage` de `CanvasEngine`, qui gère automatiquement la découpe.

## Cas particulier

Le Spritesheet peut avoir d'autres éléments graphiques qui ne respectent pas le principe de la grille à cause de leur taille.

Dans ce cas, vous pouvez définir plus précisément les zones à récupérer sur l'image :

### Zones spécifiques sur l'image

```
var spritesheet = canvas.Spritesheet.new("buttons", {  
    btn_play: [286, 27, 141, 110],  
    btn_bonus: [286, 142, 141, 60]  
});
```

Nous donnons des identifiants **1** à des zones définies par leurs positions X **2** et Y **3** et leurs dimensions (largeur **4** et hauteur **5**) dans le tableau.



# 4

## Animer les éléments sur l'écran

---

Les animations sont indispensables pour donner du dynamisme aux éléments présents sur l'écran. Ce chapitre explique comment les réaliser.

Les animations rendent le jeu très dynamique. Elles se réalisent en fonction des actions du joueur ou dans un contexte particulier, pour éviter d'avoir seulement des images figées. Nous pouvons distinguer différents types d'animations :

- les déformations (agrandir/rétrécir, faire tourner un élément) ;
- les automatismes répétitifs ;
- les réactions à une action du joueur ;
- les animations temporaires (afficher une explosion pendant un court temps).

Toutes les animations s'affichent sur la scène. Il est inutile de prendre les données vers le modèle ou le serveur (pour le jeu multijoueur).

Dans un premier temps, il faut ajouter la classe des animations lors de l'initialisation du canvas :

#### Ajouter la classe des animations

```
var canvas = CE.defines(["canvas_id"]).
    extend(Animation).
    ready(function(ctx) {
});
```

Vous pouvez maintenant utiliser les méthodes de la classe `canvas.Animation`.

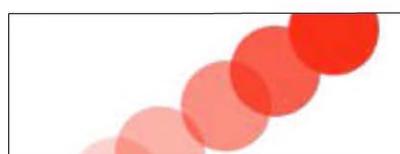
## Déformer pour animer

Ces animations déplacent des éléments sur la scène pour rendre dynamiques des transitions ou des interactions.

Le framework dispose d'une ligne temporelle virtuelle nommée `Timeline` pour changer des propriétés de l'élément dans le temps.

**Figure 4-1**

Bouger et changer l'opacité d'un cercle dans le temps



#### Déplacer un élément

```
var el = this.createElement(); ①
el.drawImage("img");
el.x = 10; ④
el.y = 10; ④

canvas.Timeline.new(el)
```

```
.to ⑤({x: "100", y: "100"}, 70) ②
.call(); ③
```

Dans cet exemple, nous créons d'abord un élément ① (lorsque la scène est prête). Nous définissons une nouvelle frise temporelle en indiquant les positions à atteindre sur 70 frames ②. Nous amorçons ensuite l'animation avec la méthode `call` ③. Ici, l'image se déplacera en diagonale des positions initiales (10,10) ④ vers les positions finales (100,100) ②.

Notez que la fonction `call` peut prendre en paramètre une fonction qui se déclenchera à la fin de l'animation.

D'autres propriétés peuvent s'appliquer :

- `scaleX` : redimensionnement vertical;
- `scaleY` : redimensionnement horizontal;
- `opacity` : opacité;
- `rotation` : rotation en radian.

La méthode `to` ⑤ peut accepter un autre paramètre pour des effets sympathiques : accélération, rebond, effet élastique, etc. Ajoutez, pour cela, des méthodes de la classe `Ease` :

#### Exemple de l'utilisation de Ease

```
canvas.Timeline.new(el)
.to({x: "100", y: "100"}, 70, Ease.easeOutQuart)
.call();
```

#### RAPPEL Pensons au tactile pour l'interaction

Rappelons que le jeu sera jouable sur les smartphones et tablettes. Les interactions étant tactiles, il faut penser à créer des boutons suffisamment larges pour les doigts.

Dans un cas pratique, nous utilisons ce type d'animation pour les interfaces présentes sur l'écran.

## Animation en boucle

Nous souhaitons afficher en boucle l'animation d'une pièce de monnaie qui tourne. Les séquences sont réparties sur une seule image dans l'ordre chronologique de gauche à droite (voir figure suivante). Chaque séquence prend donc un identifiant. On ne tient pas compte des positions.

**Figure 4-2**  
Une animation d'une pièce qui tourne



Utilisons la méthode `new` de la classe `Animation` :

#### Création d'une animation

```
var animation = canvas.Animation.new({
    images: "coin",                                // identifiant de l'image
    ➔ préchargée
    animations: {                                    // déclaration des différentes
        ➔ animations
            standard: {
                frames : [0, 5],                      // taille des séquences en pixels
                size: {
                    width: 42,
                    height: 42
                },
                fréquence: 3                         // fréquence de rafraîchissement de
            }
        ➔ l'animation
    }
});
```

- Ici, l'animation se nomme `standard` puisque c'est l'animation par défaut. Bien sûr, vous pouvez donner un nom personnalisé. Si c'est un personnage qui bouge, des noms comme `run` ou `move` sont appropriés.
- Le tableau contient deux nombres entiers. Ils correspondent respectivement à la première et à la dernière séquences. Seul l'intervalle entre ces séquences sera joué.
- Après la déclaration des propriétés, jouons en boucle l'animation sur un élément.

#### Affichage en boucle de l'animation

```
var el = this.createElement();
animation.add(el);
animation.play("standard", "loop");
```

Le code se trouve dans la méthode `ready` de la scène. Comme d'habitude, nous créons un élément et nous appliquons l'animation en le désignant dans le premier paramètre de la méthode `play`. Deux paramètres sont obligatoires :

- le nom de l'animation à jouer (ici, c'est `standard`);
- l'indication de jouer l'animation en boucle ou pas. Omettez le paramètre pour une simple animation.

### SANS FRAMEWORK Concevoir une animation

Nous créons une classe `Animation` qui lira en boucle une image avec une fréquence prédéfinie :

```
function Animation(ctx, image, animation) {
    this._images = image;
    this._animations = animation;
    this.ctx = ctx;
    this.loop(); ①
}

Animation.prototype = {
    _stop: true,
    _seq: null,
    loop: function() {
        var self = this,
            i = 0,
            freq = 0;
        function _loop() {
            var seq = self._animations[self._seq];
            if (self._stop) {
                return; ③
            }
            freq++; ④
            if (!seq.frequency) seq.frequency = 0;
            if (freq >= seq.frequency) { ⑤
                var img = Materials.images[self._images], sx = 0, sy = 0,
                    id = seq.frames[0] + i; ⑥
                if (id >= seq.frames[1]) {
                    return;
                }
                sy = parseInt(id / Math.round(img.width / seq.size.width)); ⑦
                sx = (id % Math.round(img.width / seq.size.width)); ⑧
                var w = seq.size.width * sx,
                    h = seq.size.height * sy;
                self.ctx.drawImage(self._images, w, h, seq.size.width, seq.size.
                    height, 0, 0, seq.size.width, seq.size.height);
                i++;
                freq = 0;
            }
            requestAnimationFrame(_loop);
        }
        requestAnimationFrame(_loop); ②
    },
    stop: function() {
        this._stop = true;
    },
}
```

```

    play: function(seq) {
        this._seq = seq;
        this._stop = false;
    }
});

```

Dès le départ, dans le constructeur, nous appelons la méthode `loop` ❶ qui exécute en boucle l'animation ❷. Bien entendu, puisque nous n'avons pas indiqué à l'animation de se jouer automatiquement, nous utilisons la propriété `_stop` qui n'exécute pas le code qui suit ❸.

Pour éviter une animation trop rapide, nous incrémentons une variable `freq` ❹. Lorsque cette dernière atteint la valeur de la propriété `frequence` de la séquence ❺ – qui régule le temps de rafraîchissement - la prochaine image ❻ dans les séquences est affichée dans le `canvas` comme la projection cinématographique.

Nous connaissons la taille de la séquence, mais quelle est sa position dans l'image ? Les variables `sx` ❻ et `sy` ❼ calculent les positions X et Y à l'aide de l'identifiant et de la largeur de l'image.

Pour utiliser les animations, voici la procédure : nous instancions une variable de la classe `Animation` en précisant le contexte, l'image et le nom de l'animation.

```

var canvas = document.getElementById("canvas_id"),
    ctx = canvas.getContext("2d");
var animation = new Animation(ctx, "coin", {
    standard: {
        frames : [0, 5],
        size: {
            width: 42,
            height: 42
        },
        frequence: 3
    }
});
animation.play("standard");

```

#### PRÉCISION Appels multiples à `drawImage`

Les multiples appels à la méthode `drawImage` ne vont pas afficher une « traînée » d'images car il y a un effacement à chaque frame.

## Animer en réaction à une action

Dans la section précédente, nous avons vu comment animer un élément dès que la scène est prête. Cependant, plusieurs actions sont effectuées seulement lors d'une interaction avec le joueur : appui sur une touche du clavier ou bien clic sur l'élément.

### Redimensionnement animé au passage de la souris

```
var btn = this.createElement(30, 100),
    timeline_over = canvas.Timeline.new(btn).to({scaleX: 1.5, scaleY: 1.5},
➥ 40, Ease.easeOutElastic), ①
    timeline_out = canvas.Timeline.new(btn).to({scaleX: 1, scaleY: 1}, 40,
➥ Ease.easeOutElastic); ①

btn.on("mouseover", function() {
    timeline_over.call(); ②
});

btn.on("mouseout", function() {
    timeline_out.call(); ③
});

stage.append(btn);
```

Deux lignes temporelles sur 40 frames sont créées pour redimensionner un élément ①. Nous les déclarons à l'extérieur des événements de la souris pour créer un seul écouteur.

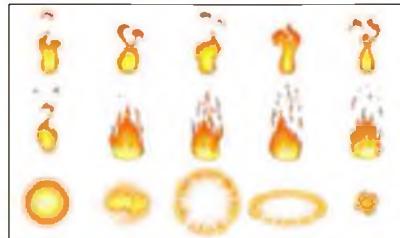
L'événement `mouseover` se déclenche lorsque le joueur passe la souris sur l'élément, qui grossit alors de 50 % avec un effet élastique ②.

L'événement `mouseout` redonne ses dimensions exactes à l'élément quand le joueur sort la souris de l'élément ③.

## Animation temporaire

L'animation s'affiche une seule fois, lors d'un événement précis dans le jeu. Lorsqu'elle se termine, elle disparaît. Nous allons voir comment créer une animation avec des propriétés différentes à chaque séquence. Prenons l'image suivante :

**Figure 4-3**  
Animation du feu



Chaque séquence est composée de motifs ordonnés pour effectuer une animation. Cependant, comme vous pouvez le constater, la dernière ligne ne s'harmonise pas avec le reste. Nous allons donc personnaliser l'animation en piochant des séquences dans un ordre différent de leur disposition initiale.

### Création d'une animation personnalisée

```
canvas.Animation.new({
    images: "Heal5",
    addIn: stage, ④
    animations: {
        standard: {
            frames: [
                [{pattern: 10, zoom: 50}], ①
                [{pattern: 10, zoom: 80}, {pattern: 1, zoom: 20}], ②
                [{pattern: 10, zoom: 100}, {pattern: 2, zoom: 120}], ③
                [{pattern: 10, zoom: 120, opacity: 200}, {pattern: 3, zoom: 120}],
                [{pattern: 10, zoom: 125, opacity: 170}, {pattern: 4, zoom: 120}],
                [{pattern: 10, zoom: 100, opacity: 150}, {pattern: 5, zoom: 120}],
                [{pattern: 10, zoom: 125, opacity: 170}, {pattern: 6, zoom: 120}],
                [{pattern: 10, zoom: 125, opacity: 200}, {pattern: 7, zoom: 120}],
                [{pattern: 10, zoom: 125, opacity: 200}, {pattern: 7, zoom: 120}],
                [{pattern: 10, zoom: 125, opacity: 170}, {pattern: 8, zoom: 120}],
                [{pattern: 10, zoom: 100, opacity: 150}, {pattern: 9, zoom: 120}],
                [{pattern: 10, zoom: 60, opacity: 180}],
                [{pattern: 10, zoom: 50, opacity: 150}],
                [{pattern: 10, zoom: 50, opacity: 100}],
                [{pattern: 10, zoom: 50, opacity: 50}]
            ],
            size: {
                width: 192,
                height: 192
            }
        }
    }
});
```

Les paramètres vous sont familiers. En revanche, `frames` est un tableau où le pointeur correspond à un temps en frames et où les éléments sont des tableaux de séquences :

**Tableau 4-1.** Début du tableau de l'exemple

Position dans le tableau	Nombre de frames total	Séquence	Propriétés
0 ①	0	10	Zoom 50
1 ②	60	10 et 1	Zoom 80 et 20
2 ③	120	10 et 2	Zoom 100 et 120

Une frame peut donc avoir plusieurs séquences afin de réaliser des combinaisons dans l'animation. `pattern` désigne la séquence (à partir de 0) et est caractérisée par des propriétés :

- `zoom` : redimensionnement (`scaleX` et `scaleY` redimensionnés proportionnellement);

- `opacity` : opacité;
- `x` : décalage sur l'axe X;
- `y` : décalage sur l'axe Y.
- La configuration de l'animation possède une propriété `addIn` ④, qui crée un élément automatiquement et le rattache à un élément parent (ici, c'est `stage`).

Nous n'avons plus qu'à jouer l'animation en spécifiant un affichage temporaire :

#### Animation temporaire

```
animation.play("standard", "remove");
```

Ici, `standard` correspond à la clé définie dans l'objet `animations` plus haut. Le dernier paramètre indique la suppression de l'élément à la fin de l'animation.



# 5

## Concevoir le Gameplay

---

Le mouvement et la gestion des entrées font partie du Gameplay. Apprenez les différents moyens de les mettre en œuvre.

Le Gameplay est très important dans un jeu vidéo et est différent selon le type du jeu. La jouabilité, combinée à une difficulté croissante, donnera une bonne expérience du jeu.

Pour concevoir le Gameplay, nous devons distinguer plusieurs parties de création :

- mouvement ;
- accélération, décélération ;
- contrôle avec le clavier, la souris, l'accéléromètre et la manette.

#### VOCABULAIRE **Gameplay**

Le Gameplay est une combinaison entre la jouabilité, la maniabilité et la difficulté donnant une expérience de jeu agréable au joueur. Il varie beaucoup selon le type de jeu : un jeu d'action recherche l'agilité et la dextérité du joueur, contrairement au jeu d'aventure plus basé sur l'exploration d'un univers.

## Mouvement

Le plus souvent, le mouvement est contrôlé par le joueur. Voyons comment affecter un objet sur l'écran avec un mouvement.

### État d'un élément

L'élément est caractérisé par un état. Selon ce dernier, ses positions vont changer. Bien sûr, s'il doit aller à un point éloigné de la carte, il faut le déplacer progressivement jusqu'à son nouvel emplacement.

Pour un adversaire, son état changera selon sa détection du personnage principal. L'intelligence artificielle déterminera son déplacement, sa vitesse ainsi que son animation.

Pour des objets ou des plates-formes, le mouvement sera aléatoire ou selon un chemin prédéfini. L'état est donc neutre et constant.

Dans tous les cas, c'est dans le rendu de la scène qu'il faut changer les positions des éléments.

### Exemple : plate-forme mobile

Nous avons une image représentant une plate-forme mobile. Elle se nomme `platform.png` et se trouve dans le dossier `images` du jeu.

**Figure 5-1**

L'image de la plate-forme mobile



**Figure 5-2**

Représentation de la plate-forme mobile dans un jeu



Le code qui suit ne comporte pas tout le contenu de la page, mais juste le modèle nommé `Game_Entity` et la scène où il est utilisé.

La plate-forme aura un mouvement linéaire et répétitif sur l'abscisse entre les positions 50 et 100 pixels. Sa direction s'inverse lorsque la plate-forme atteint une extrémité du décor.

Le mouvement de l'entité est caractérisé par plusieurs propriétés :

- `speed` : sa vitesse;
- `x` : sa position X;
- `y` : sa position Y;
- `fréquence` : sa fréquence d'affichage pour doser la vitesse du mouvement;
- `dir` : sa direction (droite ou gauche).

#### Création d'une plate-forme mobile

```
Class.create("Game_Entity", {
    speed: 1,
    x: 0,
    y: 0,
    fréquence: 2,
    currentFréquence: 0,
    dir: "right",
    initialize : function(x, y) {
        this.x = x;
        this.y = y;
    },
    moveLoop: function(a, b) {
        var new_x = this.x;
        // Fréquence de rafraîchissement
        if (this.currentFréquence >= this.fréquence) { ⑦
            if (new_x >= b) { ⑨
```

```

        this.dir = "left";
    }
    else if (new_x <= a) { ❹
        this.dir = "right";
    }
    // Application de la vitesse selon la direction
    switch (this.dir) {
        case "left":
            new_x -= this.speed; ❺
            break;
        case "right":
            new_x += this.speed; ❺
            break;
    }
    this.x = new_x;
    this.currentFrequence = 0; ❻
}
this.currentFrequence++; ❻
return new_x;
});
};

canvas.Scene.new({
    name: "map",
    materials: {
        images: {
            img: "images/platform.png"
        }
    },
    ready: function(stage) {
        // Création d'une nouvelle entité
        this.entity = Class.new("Game_Entity", [50, 50]); ❶

        // Création de l'élément avec les propriétés de l'entité
        this.el = this.createElement();
        this.el.drawImage("img"); ❷
        this.el.x = this.entity.x; ❸
        this.el.y = this.entity.y; ❸

        stage.append(this.el); ❹
    },
    render: function(stage) {

        // Déplacement en boucle entre l'intervalle [50, 100] des
        positions X
        this.el.x = this.entity.moveLoop(50, 100); ❽
        stage.refresh();
    }
});

```

Dans la méthode `ready`, nous créons une nouvelle entité dans une instance de la scène ①. Nous application l'image ② et ses positions de départ ③ et nous l'ajoutons à la scène ④.

Les propriétés dans le modèle ont des valeurs par défaut. La plate forme est donc aux positions (50,50) initialisées lors de la création du modèle.

La méthode `render` repositionne l'élément sur l'abscisse retournée par la méthode `moveLoop` du modèle ⑤. Cette méthode comporte deux paramètres pour spécifier l'intervalle où la plate-forme doit bouger.

Dans la méthode `moveLoop` du modèle, nous limitons la vitesse en ajoutant une notion de fréquence. La propriété `currentFrequency`, utilisée comme compteur, est incrémentée ⑥ à chaque rendu de la scène pour atteindre la même valeur de la propriété `frequency` ⑦. Lorsqu'elle y arrive, le calcul de la nouvelle position s'effectue et elle est remise à 0 pour recommencer la procédure ⑧.

Le calcul pour la nouvelle position X est très simple. Nous vérifions par une condition si la position X courante touche ou dépasse les extrémités imposées en paramètre ⑨. Dans ce cas, la direction de la plate-forme change et incrémente ou décrémente la vitesse à la position X ⑩.

Cette nouvelle position est affectée à la propriété pour la garder en mémoire et renvoyée à la scène pour l'affichage.

## Contrôle du joueur

La gestion du contrôle du joueur se fait avec une classe de CanvasEngine nommée `Input`. Comme pour les animations, ajoutez-la lors de l'initialisation :

### Initialisation avec la classe `Input`

```
var canvas = CE.defines(["canvas_1"]).
    extend(Input);
ready(function(ctx) {
});
```

La classe gère plusieurs entrées :

- clavier;
- accéléromètre;
- manette.

Dans tous les cas, le joueur contrôle un élément. Il faut créer un modèle approprié :

### Modèle pour la gestion du joueur

```
Class.create("Game_Player", {
    speed: 3,
    x: 0,
```

```

y: 0,
dir: "right",
initialize : function(x, y) {
    this.x = x;
    this.y = y;
},
move: function(dir) {
    this.dir = dir;
    switch (dir) {
        case "up":
            this.y -= this.speed;
            break;
        case "bottom":
            this.y += this.speed;
            break;
        case "left":
            this.x -= this.speed;
            break;
        case "right":
            this.x += this.speed;
            break;
    }
    return dir == "up" || dir == "bottom" ? this.y : this.x
}
});

```

Ce modèle gère le calcul pour le déplacement. Ici, c'est seulement un changement de position selon la direction. Cependant, il est envisageable d'ajouter des vérifications comme les collisions. Bien sûr, cela dépend de votre jeu. Nous verrons cet aspect dans le prochain chapitre.

La classe possède des propriétés habituelles (vitesse, positions et direction). La méthode `move` change les valeurs des positions : selon la direction demandée, nous renvoyons à la scène soit la position Y, soit la position X.

Le modèle `Game_Player` est utilisé dans la scène, dans la méthode `ready` pour l'initialisation :

#### **Création de l'instance player**

```
this.player = Class.new("Game_Player", [50, 50]);
```

Le joueur est placé aux positions (50, 50). Comme pour les mouvements de la plate-forme, vous pouvez aisément récupérer les propriétés du modèle pour l'affichage :

#### **Affichage d'un élément contrôlé par le joueur**

```
this.el = this.createElement();
this.el.drawImage("img");
this.el.x = this.player.x;
this.el.y = this.player.y;
```

## Clavier

Pour savoir si le joueur appuie sur une touche, utilisez la méthode `keyDown` :

### Savoir si le joueur a appuyé sur la touche Bas

```
canvas.Input.keyDown(Input.Bottom, function(e) {
    // Touche Bas enfoncee
});
```

Les touches sont représentées par des constantes pour une meilleure lisibilité du code. Ici, `Input.Bottom` est en réalité un nombre. Voici quelques constantes fréquemment utiles pour les jeux. Référez-vous à la documentation pour les autres touches :

- `Input.Bottom` : touche *Bas*;
- `Input.Up` : touche *Haut*;
- `Input.Left` : touche *Gauche*;
- `Input.Right` : touche *Droite*;
- `Input.Enter` : touche *Entrée*;
- `Input.Space` : touche *Espace*;
- `Input.Echap` : touche *Echap*;
- `Input.A` : touche *A*;
- `Input.Z` : touche *Z*.

### DIFFÉRENCE `keyPress` et `keyDown`

CanvasEngine propose deux méthodes pour l'appui sur une touche :

- `keyPress` : la fonction s'exécute une seule fois.
- `keyDown` : la fonction s'exécute tant que la touche est enfoncee.
- Notez que `keyDown` a une latence au premier appui. Pour éviter ce problème, nous effectuons le déplacement dans le rendu (voir plus loin) et non dans la fonction de rappel de cet événement

La méthode ajoute un écouteur. Cela signifie qu'il est possible d'insérer plusieurs fonctions pour une touche.

Pour faire bouger un personnage, il est inutile d'utiliser les fonctions de rappel, car vous obtiendriez un mouvement saccadé pour les deux premiers pas du personnage ; déclarez seulement les touches utilisables dans la méthode `ready` de la scène :

### Déclaration des touches utilisables

```
canvas.Input.keyDown(Input.Bottom);
canvas.Input.keyDown(Input.Up);
```

Dans la méthode `render` appelée en boucle, utilisons la méthode `isPressed` pour savoir si la touche est enfoncee par le joueur :

### Bouger si une touche est enfoncée

```
var input = { ❶
    "bottom": [Input.Bottom, "y"],
    "up": [Input.Up, "y"],
    "left": [Input.Left, "x"],
    "right": [Input.Right, "x"]
};

for (var key in input) {
    if (canvas.Input.isPressed(input[key][0])) {
        this.el[input[key][1]] = this.player.move(key); ❷
    }
}
```

La variable `input` définit les touches ❶. Chaque clé possède un tableau de deux éléments :

- la constante représentant la touche du clavier;
- la propriété à modifier (`x` ou `y`).

La boucle récupère la variable `input` pour vérifier si les touches sont enfoncées. Si c'est le cas, nous appelons la méthode `move` du modèle `Game_Player` (code plus haut) avec la direction en paramètre ❷.

#### SANS FRAMEWORK Gestion du clavier

`document` a une méthode `onkeydown` qui se déclenche lorsque une touche est enfoncée :

```
document.onkeydown = function(e) {
    if (e.which == 13) {

    }
};
```

Pour connaître la touche, testez la valeur `which` de l'événement. Ici, la valeur `13` correspond à la touche `Entrée`. Si vous voulez ajouter plusieurs écouteurs pour un type, utilisez la méthode `addEventListener` :

```
document.addEventListener("keydown", function(e) {
    if (e.which == 13) {

    }
}, false);
```

La procédure est identique avec le type `keyup` quand la touche est relâchée.

## Souris

Le joueur joue avec la souris en sélectionnant des éléments dans le jeu. Sur les tablettes tactiles et smartphones, l'appui avec le doigt revient à effectuer un clic.

Nous avons déjà parlé de l'utilisation de la souris dans le chapitre sur les écrans titres. Dans la méthode `ready` de la scène, définissons un élément :

#### Clic sur un bouton

```
var btn = this.createElement(100, 50);
btn.on("click" ①, function(e) { ②
    console.log("clic");
});
stage.append(btn);
```

Pensez bien à définir les dimensions du bouton lors de la création de l'élément et, dans la méthode `on`, le nom de l'événement **①** et la fonction de rappel **②**.

## Accéléromètre

L'accéléromètre permet de mesurer l'accélération de l'appareil. Il est souvent utilisé sur les tablettes tactiles et smartphones.

### TECHNOLOGIE Accéléromètre

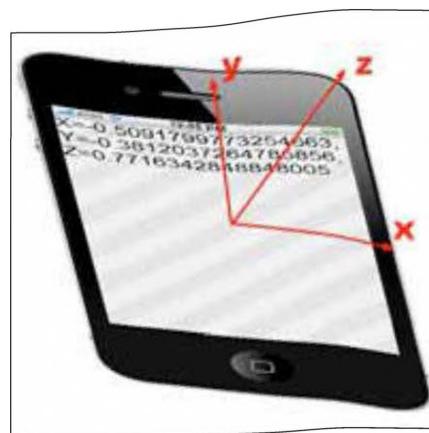
L'accéléromètre a été popularisé par les tablettes tactiles et smartphones, mais il existait depuis long-temps. Avant d'être utilisé dans les jeux, il servait par exemple pour bloquer un disque dur lorsqu'un mouvement de l'ordinateur était détecté.

Trois valeurs sont renvoyées lorsque l'appareil bouge :

Tableau 5-1. Valeur des orientations de l'appareil

Valeur	Firefox	Explication
alpha	z	Orientation horizontale avant-arrière
beta	x	Orientation horizontale gauche-droite
gamma	y	Orientation verticale haut-bas

**Figure 5-3**  
Orientations  
sur un smartphone



La méthode `ready` de la scène initialise l'écouteur de l'accéléromètre :

#### Mouvement d'un élément selon l'orientation de l'appareil

```
ready: function(stage) {
    var element = this.createElement();
    element.drawImage("img_id");
    stage.append(this.element);

    var text = this.createElement();
    text.font = 'bold 12px Arial';
    text.fillStyle = '#000';

    stage.append(text);

    canvas.Input.accelerometer(function(x, y, z) {
        element.x = -y;
        element.y = z;
        text.fillText(x + ", " + y + ", " + z, 50, 100);
    });
},
render: function {
    stage.refresh();
}
```

Un élément (représentant une bille, par exemple) se déplace sur la scène selon l'orientation de l'appareil. Le texte indique les valeurs de `x`, `y` et `z`.

**SANS FRAMEWORK Device Orientation**

Selon les plates-formes, les écouteurs varient. CanvasEngine les intègre directement dans la classe `Input`. L'écouteur est `deviceorientation` pour capturer l'orientation de l'appareil :

```
window.addEventListener('deviceorientation', function(eventData) {  
    console.log(eventData.alpha, eventData.beta, eventData.gamma);  
, false);
```

Sur Firefox, l'écouteur se nomme `MozOrientation` :

```
window.addEventListener('MozOrientation', function(eventData) {  
    console.log(eventData.x, eventData.y, eventData.z);  
, false);
```

Cependant, l'API est disponible actuellement seulement sur Firefox 12+, Google Chrome et Safari 4.2+ (iOS).

**Manette**

La manipulation de la manette dans un navigateur est très récente. Seuls Google Chrome 21+ et Firefox Nightly Build implémentent cette spécification. Il est donc important de privilégier les autres types de jouabilité. La manette doit être complémentaire.

**FIREFOX Nightly Build**

Firefox Nightly Build est une version compilée de Firefox, distribuée chaque nuit. Les développeurs peuvent ainsi suivre le développement quotidien du navigateur et tester les nouvelles fonctionnalités pour les rendre stables ultérieurement.

**Type des manettes**

Toutes les manettes ne sont pas compatibles. Sur Windows XP, Vista, 7 et 8, elles doivent reconnaître l'API `XInput`. En voici quelques-unes :

- Xbox 360 Controller
- Logitech Gamepad F310
- Logitech Rumble Gamepad F510
- Logitech Wireless Gamepad F710
- MadCatz Xbox 360 Controller

La manette de Xbox 360 fonctionne aussi sur les distributions Linux et sur Mac OS X.

**MANETTE Xbox 360**

Pour un fonctionnement sur ordinateur, vous devez vous munir d'une manette Xbox 360 filaire. Selon votre machine, il peut être indispensable d'installer les pilotes avant son utilisation. Normalement, un CD-ROM est fourni lors de l'achat de la manette pour une installation optimale.

## Interface de la manette

Figure 5-4

Interface de la Xbox 360



La manette dispose de plusieurs boutons et joysticks. Ces derniers ont un identifiant et une valeur numérique (entre -1 et 1 pour les joysticks et 0 et 1 pour les boutons).

## Utilisation de la manette

CanvasEngine utilise un framework nommé `Gamepad.js` qui reconnaît plusieurs manettes (la liste plus haut est tirée du site) et propose une petite API pour tester les touches à l'aide d'un test permanent en boucle (aucune fonction de rappel n'est déclenchée à l'appui d'un bouton). Cela peut être utile dans certains cas, mais aussi contraignant. CanvasEngine ajoute donc cette notion d'écouteurs comme pour le clavier.

► <http://www.gamepadjs.com>

## Insertion de la bibliothèque

Insérez le framework dans la page :

### Insertion du framework

```
<script src="libs/gamepad.js"></script>
```

## Test de la compatibilité de la manette

Si le joueur connecte une manette non compatible, il faut pouvoir l'en avertir. Testez avec la propriété `supported` :

### Manette reconnue ?

```
if (Gamepad.supported) {  
    // Code  
}
```

## Test de la connexion de la manette

Dans la méthode `ready` de la scène, nous initialisons la manette :

### Initialisation de la manette

```
this.gamepad = canvas.Input.Gamepad.init(function() {
    console.log("Manette connectée");
}, function() {
    console.log("Manette déconnectée");
});
```

Les deux paramètres sont facultatifs mais permettent de déclencher une fonction quand la manette est connectée et une autre lorsqu'elle est déconnectée.

### Écouteurs des boutons

Nous allons affecter un écouteur sur un bouton :

```
this.gamepad.addEventListener("faceButton0", function() {
    console.log("Bouton A enfoncé");
}, function() {
    console.log("Bouton A lâché");
});
```

Le premier paramètre est l'identifiant du bouton. Ici, `faceButton0` représente le bouton A. Deux fonctions interviennent ensuite :

- L'une se déclenche dès que le bouton est enfoncé.
- L'autre se déclenche dès le relâchement du bouton.

Les noms des boutons viennent de l'API de Gamepad.js. En voici la liste complète :

**Tableau 5-2. Liste des boutons de la manette**

Identifiant	Bouton
faceButton0	A
faceButton1	B
faceButton2	Y
faceButton3	X
leftShoulder0	LB
rightShoulder0	RB
leftShoulder1	LT
rightShoulder1	RT
select	Back
start	Start
leftStickButton	Joystick gauche appuyé
rightStickButton	Joystick droit appuyé

dpadUp	Haut des touches directionnelles
dpadDown	Bas des touches directionnelles
dpadLeft	Gauche des touches directionnelles
dpadRight	Droite des touches directionnelles

Si des écouteurs sont déjà affectés aux touches du clavier, il est fastidieux de devoir répéter des opérations. Un raccourci dans CanvasEngine permet de déclencher une fonction d'un écouteur du clavier :

#### Attacher la même fonction de la touche Droite du clavier à la manette

```
canvas.Input_KeyDown(Input.Right, function() {
    // Code
});
this.gamepad = canvas.Input.Gamepad.init();
this.gamepad.addEventListener("dpadRight", Input.Right);
```

Ici, si le joueur appuie sur la touche *Droite* du clavier, la fonction de l'écouteur sera déclenchée pour, par exemple, faire bouger son personnage. Le bouton *Droite* de la manette fera de même.

#### Mise à jour des entrées de la manette

Pour le moment, nous avons uniquement créé des écouteurs. Mettons à jour les entrées de la manette. Pour cela, ajoutez la ligne de code suivante dans la méthode `render` :

#### Mise à jour des entrées de la manette

```
this.gamepad.update();
```

Dans cette méthode, si vous voulez tester directement la valeur d'une touche, procédez de la manière suivante :

```
var pad = Gamepad.getStates()[0];
if (pad) {
    if (pad.faceButton0) {
        // Code
    }
}
```

Nous récupérons les états de la première manette (position 0 dans le tableau renvoyé par la méthode `getStates`). Si la manette existe, nous testons la valeur de `faceButton0`. La valeur est 1 si le bouton est enfoncé, 0 sinon. Le code à l'intérieur de la condition sera exécuté en boucle tant que le bouton *A* est enfoncé.

Pour le déplacement des joysticks, le procédé est équivalent, sauf que la valeur est comprise entre -1 et 1. La valeur 0 indique que le joystick est à sa position initiale. Plus le

joystick est tiré dans ses extrémités, plus sa valeur s'approche de 1 ou -1 pour l'autre sens. Les identifiants sont les suivants :

Tableau 5-3. Liste des identifiants du Joystick

Identifiant	Joystick
leftStickX	Joystick gauche sur l'axe X
leftStickY	Joystick gauche sur l'axe Y
rightStickX	Joystick droit sur l'axe X
rightStickY	Joystick droit sur l'axe Y

Prenons un exemple. Nous souhaitons tester le pourcentage du joystick gauche tiré sur l'axe X :

#### Tester le joystick gauche sur l'axe X

```
var pad = Gamepad.getStates()[0];
if (pad) {
    if (pad.leftStickX < -0.2 || pad.leftStickX > 0.2) {
        if (pad.leftStickX > 0) {
            console.log("Droite", Math.abs(pad.leftStickX * 100) + "%");
        }
        else {
            console.log("Gauche", Math.abs(pad.leftStickX * 100) + "%");
        }
    }
}
```

Puisque le joystick est sensible, nous testons une condition pour n'avoir que les valeurs inférieures à 20 % ou supérieurs à 20 %. Si la valeur s'approche de 1, cela signifie que le joueur titre le joystick sur la droite. Si la valeur tend vers -1, le joystick part sur la gauche.

## Accélération et décélération

L'accélération consiste à augmenter progressivement la vitesse. La décélération est la démarche inverse. Nous retrouvons souvent ces mouvements dans les jeux de type plate-forme.

**Figure 5-5**

Décélération sur la glace pour donner la sensation d'un glissement  
(Rayman Origins)



Rappelons que tous les calculs se font sur le modèle. La méthode `render` de la scène ne sert qu'à afficher les éléments.

## Accélération

L'accélération s'effectue dès le départ du mouvement, c'est-à-dire lorsque le joueur appuie sur une touche. Un facteur multiplicatif `a`, initialisé à 0, augmente à chaque rendu pour augmenter la vitesse par relation de cause à effet.

### Propriété «`a`» utilisée comme facteur

```
Class.create("Game_Player", {
    speed: 4,
    x: 0,
    y: 0,
    a: 0
});
```

Reprendons la méthode `move` du début du chapitre : en voici le code modifié :

### Méthode `move` sur l'axe X seulement

```
move: function(dir) {
    this.dir = dir;
    this.a += .05;
    if (this.a >= 1) {
        this.a = 1;
    }
    var speed = this.speed * this.a;
    switch (dir) {
        case "left":
            this.x -= speed;
            break;
        case "right":
            this.x += speed;
            break;
    }
    return this.x;
}
```

Le principe du mouvement ne change pas. Ici, nous restons sur l'axe X puisque l'axe Y est réservé à la gravité. À chaque fois que la méthode `move` est appelée, la propriété `a` est bien augmentée pour la multiplier à la vitesse. La première condition limite l'accélération afin que le mouvement du joueur puisse devenir stable.

La propriété de l'accélération gardera sa valeur lors du deuxième mouvement de la part du joueur. L'accélération ne recommencera pas. Pour résoudre ce problème, nous devons réinitialiser la propriété à 0 lorsque la touche est relâchée.

Dans un premier temps, ajoutez la méthode `moveClear` dans le modèle pour la réinitialisation :

#### Réinitialisation du facteur «a»

```
moveClear: function() {
    this.a = 0;
}
```

Dans la méthode `ready` de la scène, ajoutez des écouteurs sur les touches du clavier :

#### Écouteurs sur les touches du clavier

```
var self = this;
canvas.Input.keyUp(Input.Right, function() {
    self.player.moveClear();
    // Animation et autres codes
});
canvas.Input.keyUp(Input.Left, function() {
    self.player.moveClear();
    // Animation et autres codes
});
```

Les fonctions appellent notre méthode précédemment créée.

## Décélération

La vitesse diminue quand le joueur relâche la touche correspondant à la direction. Cela signifie que le mouvement doit tout de même continuer sur une courte durée. Comme pour l'accélération, nous devons initialiser une propriété `d`, facteur multiplicatif de la vitesse de l'élément.

#### Initialisation de la propriété d

```
Class.create("Game_Player", {
    a: 0, // Accélération
    d: 1, // Décélération
    dece_dir: false,
    // ...
});
```

La valeur initiale est `1` car elle décroîtra pour diminuer la vitesse. Une autre propriété annonce la direction de la décélération en cours. Elle vaudra `right` pour la droite, `left` pour la gauche ou le booléen `false` si aucune décélération est en cours.

Ajoutons dans le modèle la méthode `decelerationUpdate` :

#### Gestion de la décélération

```
decelerationUpdate: function() {
    var dir = this.dece_dir;
    if (dir) {
        this.d -= .05;
        if (this.d <= 0) {
            this.d = 0;
            this.dece_dir = false;
        }
        var speed = this.speed * this.d;
        switch (dir) {
            case "left":
                this.x -= speed;
                break;
            case "right":
                this.x += speed;
                break;
        }
    }
    return this.x;
}
```

Cette méthode est appelée en boucle dans le rendu de la scène pour vérifier si une décélération est en cours. Si c'est le cas, nous récupérons la direction et appliquons de nouvelles positions au joueur.

La propriété `d` fonctionne de la même manière que l'accélération mais avec une décrémentation. Arrivés à la valeur `0`, nous indiquons que la décélération est terminée en redonnant le booléen `false` à la propriété `dece_dir`.

Nous ajoutons un modificateur pour la direction de l'accélération :

#### Modificateur de la direction

```
setDeceleration: function(dir) {
    this.dece_dir = dir;
}
```

Dans la méthode `render` de la scène, nous appliquons le calcul effectué à l'élément représentant le joueur :

#### Affichage de la décélération

```
this.el.x = this.player.decelerationUpdate();
```

Nous avons déclaré au début de la section que la décélération a lieu quand le joueur relâche la touche. Nous devons ajouter des écouteurs sur les touches en question pour changer la valeur de la direction de la décélération et ainsi faire tourner la méthode `decelerationUpdate` créée plus haut :

#### Affectation de la décélération lorsque la touche est relâchée

```
var self = this;
canvas.Input.keyUp(Input.Right, function() {
    self.player.moveClear();
    self.player.setDeceleration("right");
    // Animation et autres codes
});

canvas.Input.keyUp(Input.Left, function() {
    self.player.moveClear();
    self.player.setDeceleration("left");
    // Animation et autres codes
});
```

Nous avons gardé la méthode `moveClear` pour réinitialiser aussi la propriété `d` du modèle :

#### Réinitialisations des facteurs

```
moveClear: function() {
    this.a = 0; // Accélération
    this.d = 1; // Décélération
}
```

## Gravité pour le saut

La gravité est souvent utilisée dans un jeu de plate-forme : elle tire le héros vers le bas de l'écran tant qu'aucune plate-forme n'est sur son chemin. Le joueur peut ainsi sauter de plate-forme en plate-forme en appuyant sur une touche du clavier.

**Figure 5-6**  
Saut et gravité dans  
Super Mario World



Vous devez retenir que le personnage dirigé a plusieurs états :

- en contact avec une plate-forme;
- en train de sauter;
- en train de tomber.

Ces états permettent de bloquer certaines actions ou d'en autoriser d'autres. Trois règles s'appliquent :

- 1 Le saut n'est possible que si le héros est au contact d'une plate-forme.
- 2 Si le héros tombe, il ne peut pas sauter.
- 3 Le héros ne tombe pas à cause de la gravité s'il est en train de sauter.

Nous utiliserons l'extrême basse du `canvas` pour les collisions et arrêter la gravité.

## Initialisation

### Initialisation des propriétés pour la gravité et le saut

```
Class.create("Game_Player", {
    currentState: "",
    _gravity: {
        power: 10,
        velocity: 0
    },
    _jump: {
        power: 10,
        velocity: 1
    }
//...
});
```

Trois propriétés définissent la gravité, l'état du joueur et le saut :

- `currentState` est un identifiant désignant l'état courant du joueur :
  - `platform` : au contact avec une plate-forme;
  - `jumping` : en train de sauter;
  - `godown` : en train de tomber.
- `gravity` est un objet indiquant la puissance du saut et un facteur de vitesse (vitesse dans une direction) pour la gravité.
- `_jump` est identique à la gravité mais pour le saut.

## Gravité

La méthode `gravityUpdate` que nous allons créer permet de faire tomber le personnage avec une vitesse ralentie progressivement jusqu'à ce qu'il atteigne le bas de l'écran et, bien sûr, s'il n'est pas en train de sauter :

### Application de la gravité

```
gravityUpdate: function() {
    if (this.currentState != "jumping") {
        if (this.y + this.height + this.speed <= this.map_height) {
            this._gravity.velocity += .05;
            if (this._gravity.velocity >= 1) {
                this._gravity.velocity = 1;
            }
            this.y += (this._gravity.power * this._gravity.velocity);
        }
        else {
            this.currentState = "platform";
            this._gravity.velocity = 0;
        }
    }
    return this.y;
}
```

La première condition teste l'état du joueur (règle 3) et la deuxième vérifie s'il y a collision. Les propriétés `height` (hauteur du personnage) et `speed` (vitesse) sont utilisées dans cet exemple.

Ainsi, on peut tester la prochaine position Y du coin inférieur gauche de l'élément. Si elle ne dépasse pas le bas du niveau, nous appliquons une accélération (voir la section précédente) sur la propriété `y`.

Dans le cas contraire, c'est-à-dire lorsque la prochaine position Y touche le bas du niveau, nous changeons l'état du joueur et réinitialisons à 0 la valeur de la propriété `velocity`.

Dans la méthode `render` de la scène, nous appelons constamment `gravityUpdate` pour appliquer la gravité à n'importe quel moment :

### Changement de la position Y selon la gravité

```
this.player.y = this.player.gravityUpdate();
```

## Saut

Pour le saut, le processus est le contraire de la gravité : on diminue la valeur de la position Y avec une vitesse accélérée. Dans le modèle, on ajoute :

### Application du saut

```
jumpUpdate: function() {
    if (this.currentState == "jumping") {
        this._jump.velocity -= .05
        if (this._jump.velocity <= 0) {
            this._jump.velocity = 1;
            this.currentState = "godown";
        }
        else {
            this.y -= this._jump.power * this._jump.velocity;
        }
    }
    return this.y;
}
```

La première condition vérifie que le joueur est en train de sauter pour éviter de sauter lorsqu'il tombe (règle 2). Une accélération est alors effectuée (voir section précédente).

Lorsque le personnage arrive au sommet de son saut, le statut devient `godown` pour que la gravité rentre en jeu.

La méthode `jumpUpdate` est appelée à l'enfoncement d'une touche, dans la méthode `render` de la scène :

### Faire sauter si la touche Haut est enfoncée

```
if (canvas.Input.isPressed(Input.Up)) {
    this.player.jumpUpdate();
}
```

Cependant, la méthode fonctionne seulement lorsque l'état du personnage est `jumping`. Nous devons changer l'état à l'enfoncement de la touche ainsi qu'à son relâchement. Dans le modèle, on ajoute :

### Changer l'état du saut

```
jump: function(state) {
    if (state && this.currentState == "platform") {
        this.currentState = "jumping";
    }
    else if (!state && this.currentState == "jumping") {
        this._jump.velocity = 1;
        this.currentState = "godown";
    }
}
```

Le paramètre `state` est un booléen. S'il vaut `true`, cela signifie qu'un saut est déclenché. Dans le cas contraire, le personnage doit retomber.

La première condition permet de respecter la règle 1. Le personnage peut sauter seulement s'il est sur une plate-forme.

La deuxième condition indique au personnage de retomber en cours de saut. Cela permet d'établir une intensité sur le saut selon la durée d'appui sur la touche.

Dans la méthode `ready` de la scène, nous ajoutons deux écouteurs pour changer l'état du personnage :

#### Écouteurs sur la touche Haut

```
var self = this;
canvas.Input.press(Input.Up, function() {
    self.player.jump(true);
    // Animation et autres codes
});

canvas.Input.keyUp(Input.Up, function() {
    self.player.jump(false);
    // Animation et autres codes
});
```



# 6

## Avancer le joueur avec des défilements

---

Le défilement du décor donne l'illusion que le personnage contrôlé avance sur la carte ou le niveau.

Une carte ou un niveau peut dépasser la taille de l'écran. Dans ce cas, lorsque le joueur bouge, il risque de sortir de l'écran et le joueur ne pourra plus le contrôler. C'est donc la carte qu'il faut faire défiler en arrière-plan pour donner l'illusion que le personnage se déplace. On nomme cette procédure scrolling. On a coutume de distinguer défilement classique et défilement différentiel.

## Défilement classique

Le décor se déplace à la même vitesse que le joueur. CanvasEngine propose la classe `Scrolling` pour gérer le défilement.

### Définir les éléments à déplacer

Tout d'abord, nous ajoutons la classe lors de l'initialisation du `canvas` :

#### Ajout du Scrolling

```
var canvas = CE.defines("canvas").
    extend(Scrolling).
    ready(function(ctx) {
});
```

Dans la méthode `ready` de la scène, nous ajoutons le défilement en spécifiant trois paramètres :

- la scène;
- la largeur du carreau en pixel;
- la hauteur du carreau en pixel.

#### Définir un nouveau Scrolling

```
var W_TILE = 32,
    H_TILE = 32;
this.scrolling = canvas.Scrolling.new(this, W_TILE, H_TILE);
```

L'objet `scrolling` est une instance de la classe accessible depuis la méthode `render`. Pour le moment, définissons l'élément principal qui reste centré lorsqu'il bouge (généralement, c'est le personnage principal) :

#### Ajout du héros

```
var player = this.createElement();
this.scrolling.setMainElement(player);
```

Nous procédons de la même manière pour ajouter avec la méthode `addScroll` les éléments du décor, qui doivent défiler dans le sens contraire à celui du héros.

#### Ajout du décor

```
var map = this.createElement();
var scroll_map = this.scrolling.addScroll({
    element: map,           // l'élément créé représentant le décor
    speed: 3,                // la vitesse du défilement en pixels
    block: true,              // l'élément ne défile plus quand les extrémités
    ↴ touchent le bord du canvas
```

```

    width: 120,           // largeur du niveau en pixels
    height: 50            // hauteur du niveau en pixels
});
```

## Rafraîchissement du déplacement

Dans la méthode `render` de la scène, ajoutons la ligne suivante :

### Mise à jour du Scrolling

```
this.scrolling.update();
```

Le déplacement de la carte est désormais opérationnel !

## Défilement différentiel

Le défilement différentiel est une technique pour accentuer une sensation de profondeur sur les arrière-plans. Il consiste à diminuer la vitesse de défilement par rapport à celle du personnage.

**Figure 6-1**

Défilement différentiel dans Angry Birds



Sur la figure, nous distinguons l'arrière plan (1) et le premier plan (3) avec les Sprites (2).

### Établir un défilement différentiel sur un arrière-plan

```

var W_TILE = 32,
    H_TILE = 32;
var background = this.createElement();
background.drawImage("background");
stage.append(background);

this.scrolling = canvas.Scrolling.new(this, W_TILE , H_TILE );
this.scrolling.setMainElement(player);

this.scrolling.addScroll({
  speed: 1,
```

```
    element: background,
    block: true,
    parallax: true,
    width: 120,
    height: 50
});
```

Le principe de la déclaration ne change guère. La propriété `parallax` indique à l'élément de se déplacer sans se soucier du centrage du personnage. Remarquez que la vitesse définie dans la propriété `speed` est plus faible que le défilement dans la section précédente.

# 7

## Interaction avec les objets

---

Comment faire interagir le personnage avec les éléments du décor (collisions, déclenchement d'actions) ?

Un décor peut bloquer le mouvement, déclencher un passage ou faire perdre le joueur. Tout dépend du type de jeu. Dans un jeu d'arcade où le joueur doit avancer de gauche à droite dans un niveau et éviter les murs, la collision est fatale. Dans un RPG, si un coffre est présent sur la carte, l'interaction se fait seulement à l'appui d'une touche et le héros gagnera sûrement un objet pour continuer son aventure.

Voyons comment réaliser, de façon basique et précise, des collisions, des interactions avec le décor et selon une action de la part du joueur.

## Collision

Une collision est un contact entre deux éléments. Dans un jeu, c'est particulièrement le joueur qui rentre en collision avec le décor, ce qui arrête son mouvement.

Dans la gestion des collisions, nous allons séparer notre code en trois parties :

- la scène qui insère la carte, ajoute les Sprites et déclare les entités dans le modèle ;
- les Sprites qui gèrent *seulement* l'affichage ;
- le modèle *Game\_Map* qui teste la praticabilité entre les positions du joueur et les entités.

### BONNE PRATIQUE

Rappelez-vous que cette façon de procéder améliore la structure de votre code, mais aussi une compatibilité future du multijoueur.

Le personnage a une zone d'interaction nommée *Hitbox*. Généralement, cette dernière est un rectangle placé sur la surface du Sprite.

**Figure 7-1**

Hitbox sur un personnage



Dans la pratique, c'est les quatre sommets du rectangle qui nous servent pour calculer les collisions et les interactions.

Les positions  $X_1$  et  $Y_1$  du héros correspondent au point (1). Les autres points se calculent à l'aide des dimensions du personnage :

- Point (2) : Position  $X_2 = \text{Position } X_1 + \text{Largeur}$
- Point (3) : Position  $X_3 = \text{Position } X_1 + \text{Largeur}$  et Position  $Y_3 = \text{Position } Y_1 + \text{Hauteur}$
- Point (4) : Position  $Y_4 = \text{Position } Y_1 + \text{Hauteur}$

Admettons que les propriétés se nomment `width` pour la largeur et `height` pour la hauteur, la classe *Game\_Player* se résume à :

### Classe *Game\_Player* simplifiée

```
Class.create("Game_Player", {
    speed: 4,
    x: 0,
    y: 0,
    dir: "right",
    height: 32,
    width : 32,
```

```

    initialize: function(x, y) {
        this.x = x;
        this.y = y;
    }
});
```

La classe n'est pas complète car les mouvements et autres données sont à ajouter. Pour le mouvement, reportez-vous au chapitre 5. Nous allons ajouter une condition de praticabilité dans la méthode `move`.

## Au bord de la carte

Le schéma de données (chapitre 3) détermine les bords du niveau. C'est principalement le modèle `Game_Map` qui teste si le joueur arrive au bord de la carte :

### La classe `Game_Map` pour tester la praticabilité

```

Class.create("Game_Map", {
    map: null,
    initialize: function(map) {
        this.map = map;
    },
    isPassable: function(player, new_x, new_y) {
        if (new_x < 0 || (new_x + player.width) > this.map.getWidthPixel()) {
            return false;
        }
    }
});
```

Dans la gestion des collisions, nous devons tester la prochaine position du héros. Si elle est impraticable, le calcul effectué sur le mouvement n'est pas altéré. Dans le cas contraire, le héros bouge normalement.

La propriété `map` est un objet comportant les informations de la carte. Voyons comment bloquer le mouvement du joueur. Nous vérifions les bords gauche et droite de la carte par rapport aux points (1) et (2).

### La classe `Game_Player` avec la méthode du mouvement

```

Class.create("Game_Player", {
    speed: 4,
    x: 0,
    y: 0,
    dir: "right",
    height: 32,
    width: 32,
    map: null,
    initialize: function(x, y, map) {
        this.x = x;
        this.y = y;
```

```

        this.map = map;
    },
    move: function(dir) {
        this.dir = dir,
        x = this.x;
        switch (dir) {
            case "left":
                x -= this.speed;
                break;
            case "right":
                x += this.speed;
                break;
        }
        if (this.map.isPassable(this, x, this.y)) {
            this.x = x;
        }
        return this.x;
    }
});

```

Si nous reprenons la classe `Game_Player` du chapitre 5, le personnage bouge selon une vitesse. Maintenant cependant, la variable `x` avec la nouvelle valeur sera attribuée à la propriété `x` seulement si la prochaine destination est praticable.

La scène utilise la classe `Tiled` pour récupérer plusieurs informations sur la carte :

#### La scène déclarant les modèles

```

canvas.Scene.new({
    name: "Map",
    materials: {
        images: {
            player: "player.png",
            tileset: "tileset.png"
        }
    },
    ready: function(stage) {
        var self = this;
        var tiled = canvas.Tiled.new();
        tiled.load(this, stage, "map/map.json");
        tiled.ready(function() {
            var tile_w = this.getTileWidth(),
                tile_h = this.getTileHeight();
            self.game_map = Class.new("Game_Map", [this]);
            self.game_player = Class.new("Game_Player", [8 * tile_w, 12 *
                tile_h, self.game_map]);
        })
        render: function(stage) {
        }
    });
}
);

```

## Sur le décor et les objets

Au chapitre 3 sur le Level Design, nous avons vu que les carreaux ainsi que les Sprites possèdent des propriétés comme la superposition et la praticabilité. Cette dernière nous intéresse justement pour bloquer le mouvement du personnage.

### Collision sur le décor

Tout d'abord, reprenez l'éditeur Tiled Map Editor et ajoutez une propriété nommée *passable* sur un carreau.

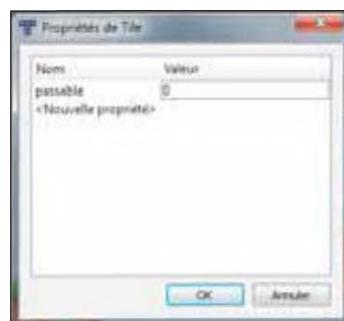
- 1 Cliquez-droit sur le carreau et choisissez *Propriétés des Tiles*.

**Figure 7-2**  
Affecter des propriétés  
au carreau



- 2 Cliquez sur <*Nouvelle propriété*>. Rentrez *passable* pour le nom et 0 comme valeur pour indiquer que le carreau est impraticable.

**Figure 7-3**  
Propriété passable  
sur le carreau



Il suffit maintenant de modifier la méthode *isPassable* dans les codes précédents :

#### Test de la praticabilité sur les carreaux

```
Class.create("Game_Map", {
    map: null,
    initialize: function(map) {
        this.map = map;
    },
    isPassable: function(player, new_x, new_y) {
        var ent, self = this;

        if (new_x < 0 && (new_x + player.width) > this.map.getWidthPixel()) {
            return false;
        }
    }
});
```

```

        }

        var tile_w = this.map.getTileWidth(),
            tile_h = this.map.getTileHeight();

        function pointIsPassableInTile(x, y) { ②
            var map_x = Math.floor(x / tile_w), ③
                map_y = Math.floor(y / tile_h); ③

            var props = self.map.getTileProperties(null, map_x, map_y); ④
            for (var i=0 ; i < props.length ; i++) { ⑤
                if (props[i] && props[i].passable == "0") { ⑥
                    return false;
                }
            }
            return true;
        }

        if (!pointIsPassableInTile(new_x, new_y) ||
            !pointIsPassableInTile(new_x + player.width, new_y) ||
            !pointIsPassableInTile(new_x, new_y + player.height) ||
            !pointIsPassableInTile(new_x + player.width, new_y + player.
        ↪ height)) {
            return false; ①
        }
        return true;
    }
});
```

Les quatre points du Sprite nécessitent un test. Si l'un d'eux ne satisfait pas les critères des conditions, nous renvoyons `false` pour indiquer au joueur que le carreau est impraticable ①. La fonction privée permet de réaliser les différents tests à plusieurs reprises ②.

- 1 Selon les positions du personnage en pixels, nous devons récupérer la position dans le tableau. La conversion reste enfantine : divisez la position par la taille du carreau ③.
- 2 La méthode `getTileProperties` de la classe `Tiled` renvoie un tableau des différentes couches avec les propriétés du carreau selon les positions dans le tableau ④.
- 3 Nous parcourons le tableau ⑤ pour, dans un premier temps, vérifier si la donnée existe et, dans un deuxième temps, observer la praticabilité du carreau selon sa propriété `passable` ⑥.

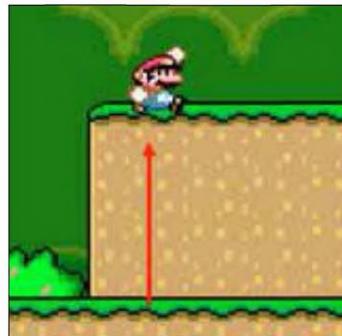
### Collision sur le décor selon une direction

Dans plusieurs jeux, un carreau peut être franchissable ou non selon la direction du personnage. Ainsi, si ce dernier se déplace vers la droite, il pourra traverser un carreau infranchissable dans la direction inverse.

Nous utilisons les combinaisons hexadécimales pour les tests des collisions selon une direction. L'intérêt majeur est de n'écrire qu'une seule condition et, par conséquent, d'éviter de multiples tests fastidieux.

**Figure 7-4**

Le personnage peut traverser la plate-forme seulement vers le haut (Super Mario World).



Chaque bit représente une direction.

**Tableau 7-1.** Valeur hexadécimale des quatre directions

Hexadécimal	Binaire	Direction
1	0001	Haut
2	0010	Droite
4	0100	Bas
8	1000	Gauche

On affecte au carreau une valeur hexadécimale cumulant les directions autorisées.

**Tableau 7-2.** Exemples de valeurs hexadécimales

Valeur du carreau	Binaire	Direction(s) autorisée(s)
8	1000	Seulement sur la gauche
C	1100	Sur la gauche et vers le bas
F	1111	Les 4 directions

Le code pour tester la praticabilité change un peu dans la fonction privée :

#### Test des directions avec l'hexadécimal

```
function pointIsPassableInTile(x, y) {
    var map_x = Math.floor(x / tile_w),
        map_y = Math.floor(y / tile_h);

    var dir_hexa = 0x0;
    switch (player.dir) {
        case "left":
            dir_hexa = 0x8; ①
            break
        case "right":
            dir_hexa = 0x2; ①
            break
    }

    var props = self.map.getTileProperties(null, map_x, map_y);
    for (var i=0 ; i < props.length ; i++) {
        if (props[i] && (parseInt(props[i].passable, 16) & dir_hexa) != ②
dir_hexa) { ②
            return false;
        }
    }
    return true;
}
```

Selon la direction du joueur (deux seulement sont représentées ici ①), nous indiquons la valeur hexadécimale correspondante. La condition binaire vérifie si le bit de la direction est présent dans la valeur du carreau ②.

#### RAPPEL Hexadécimal et condition binaire

L'hexadécimal est un système de numérotation sur une base 16 utilisant les « chiffres » suivants : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Ainsi le nombre 10 en base 10 est égal à A en base 16.

En Javascript, les nombres sont préfixés de `0x` pour indiquer la base.

L'intérêt d'utiliser une base d'une puissance de 2 est de manier les combinaisons binaires plus facilement (base 2). La condition binaire est représentée par les symboles `&` pour le « Et binaire » et `|` pour le « Ou binaire ». Dans notre cas, pour chaque bit, nous réalisons une multiplication pour avoir un résultat final :

`1010 & 1100` est égal à `1000`

`1110 & 0011` est égal à `0010`

Transformé en hexadécimal :

`0xA & 0xC` est égal à `0x8`

`0xE & 0x3` est égal à `0x2`

## Collision sur des objets

Contrairement au décor qui reste fixe, nous devons comparer les positions du héros à celles des objets présents sur la carte :

### Gestion de la collision sur une entité

```
Class.create("Game_Map", {
    entities: [],
    initialize: function() {

    },
    addEntity: function(id, data) {
        var entity = Class.new("Game_Entity", [data.x, data.y, data.width,
    ↵ data.height]);
        this.entities.push(entity);
        return entity;
    },
    isPassable: function(player, new_x, new_y) {
        var ent;

        // Test de la praticabilité du décor

        function pointIsPassable(ent, x, y) {
            if (x >= ent.x && x <= ent.x + ent.width &&
                y >= ent.y && y <= ent.y + ent.height) {
                return false;
            }
            return true;
        }

        for (var i=0 ; i < this.entities.length ; i++) {
            ent = this.entities[i];
            if (!pointIsPassable(ent, new_x, new_y) &&
                !pointIsPassable(ent, new_x + player.width, new_y) &&
                !pointIsPassable(ent, new_x, new_y + player.height) &&
                !pointIsPassable(ent, new_x + player.width, new_y + player.
    ↵ height)) {
                return false;
            }
        }
        return true;
    });
});
```

Les objets ou entités sont ajoutés avec la méthode `addEntity` dans la scène. Pour la classe des entités, nous reprenons celle présentée au chapitre 5 sur les mouvements.

Les tests des collisions ne sont pas très différents de ceux pour le décor. Nous testons si l'un des 4 points du personnage est à l'intérieur de l'entité. Si c'est le cas, nous renvoyons la valeur `false` pour empêcher au joueur de bouger.

### Affichage des entités

Définissons, pour l'exemple, une classe `Sprite_Coins` pour afficher des pièces dans le jeu.

#### Classe pour l'affichage d'un Sprite

```
Class.create("Sprite_Coins", {
    el: null,
    initialize: function(id, scene, layer, data) {
        this.id = id; ②
        this.el = scene.createElement(data.width, data.height); ①
        // Code
        this.setPosition(data.x, data.y);
        layer.append(this.el); ③
    },
    setPosition: function(x, y) {
        this.el.x = x;
        this.el.y = y;
    }
});
```

Le plus important est de créer un élément en lui attribuant une taille ①. Ces paramètres nous serviront dans la gestion des collisions.

N'oubliez pas l'identifiant ②. Il crée un lien entre les entités dans le modèle et l'affichage sur la scène car l'identifiant est identique dans les deux classes.

Enfin, la couche (paramètre `layer`) est celle des objets au même niveau que le héros du jeu ③.

#### Scène ajoutant les Sprites dans le modèle

```
canvas.Scene.new({
    name: "Map",
    materials: {
        images: {
            player: "player.png",
            tileset: "tileset.png",
            coin: "coin.png",
        }
    },
    sprites: {},
    addCoin: function(id, layer, data) {
        this.sprites[id] = Class.new("Sprite_Coins", [id, this, layer,
        data]); ③
        this.game_map.addEntity(id, data); ④
    },
    ready: function(stage) {
```

```

var self = this;
var tiled = canvas.Tiled.new();
tiled.load(this, stage, "map/map.json");

tiled.ready(function() {
    var tile_w = this.getTileWidth(),
        tile_h = this.getTileHeight(),
        layer_event;
    self.game_map = Class.new("Game_Map", [this]); ①
    self.game_player = Class.new("Game_Player", [8 * tile_w, 12 *
    ↵ tile_h, self.game_map]); ②

    self.player = self.createElement(32, 32);
    self.player.drawImage("player");
    self.player.x = self.game_player.x;
    self.player.y = self.game_player.y;

    layer_event = this.getLayerObject();

    self.addCoin(1, layer_event, {
        x: 50,
        y: 250,
        width: 32,
        height: 32
    });

    layer_event.append(self.player);
}
render: function(stage) {
    // code
}
});

```

La méthode `ready` reprend le Level Design du chapitre 3. Désormais, nous créons une instance de la classe `Game_Map` ①, décrite plus haut, injectée en troisième paramètre du constructeur de la classe `Game_Player` ② (chapitre 5).

La méthode `addCoin` effectue deux créations :

- un Sprite pour l'affichage ③ ;
- un appel dans la classe `Game_Map` pour créer une entité ④.

Dans les deux cas, l'identifiant ainsi que les positions sont identiques.

## Les entités

Voici une classe très simple qui ne fait qu'identifier la position et la taille d'une entité :

### Classe simple représentant les données d'une entité

```

Class.create("Game_Entity", {
    x: 0,

```

```

y: 0,
width: 0,
height: 0,
initialize: function(x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
});

```

## Interaction

L'interaction consiste à déclencher une action ou un événement au contact de l'élément. Nous reprenons le système de collisions en ajoutant des fonctions à appeler pour affecter les Sprites sur la scène.

**Figure 7-5**

Animation lorsque le poisson rentre en collision avec la pièce



## Déclenchement automatique au contact

Nous aurons besoin d'une propriété qui indique si l'entité est en contact ou non. Les fonctions sont exécutées selon les conditions suivantes :

- si le joueur est en contact pour la première fois;
- si le joueur n'est plus en contact (mais l'a été).

### Test de l'interaction des modèles

#### Déclaration de fonctions des contacts

```

Class.create("Game_Entity", {
    speed: 1,
    x: 0,
    y: 0,
    width: 0,
    height: 0,
    dir: "right",

```

```

    _hit: false,
    _call_hit: {},
    initialize: function(x, y, width, height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;

    },
    callHit: function(on, off) { ❶
        this._call_hit.on = on;
        this._call_hit.off = off;
    },
    hit: function(val) {
        if (val && !this._hit) { ❸
            if (this._call_hit.on) this._call_hit.on.call(this);
        }
        else if (!val && this._hit) {
            if (this._call_hit.off) this._call_hit.off.call(this);
        }

        this._hit = val;
    },
    isHit: function() {
        return this._hit;
    }
};

);

```

La méthode `callHit` ❶ attribue une fonction pour le contact et une autre lorsque ce contact est rompu. Ces fonctions seront déclenchées dans la méthode `hit` ❷. Si `val` vaut `true`, autrement dit si un contact se produit, nous appelons la fonction seulement la première fois ❸. La deuxième condition réagit de la manière inverse.

Reprendons la fonction privée `pointIsPassable` dans la classe `Game_Map`.

#### Appel des fonctions à l'interaction de l'entité

```

function pointIsPassable(ent, x, y) {
    if (x >= ent.x && x <= ent.x + ent.width &&
        y >= ent.y && y <= ent.y + ent.height) {
        ent.hit(true); ❹
        return false;
    }
    ent.hit(false); ❹
    return true;
}

```

Nous appelons la méthode `hit` de l'entité en signalant si le joueur est au contact ou non.

### Affichage de l'interaction

Après le calcul et la vérification des interactions, nous devons les afficher sur un élément. Pour cela, reprenons la classe `Sprite_Coins`.

#### Affichage d'une animation au contact

```
Class.create("Sprite_Coins", {
    el: null,
    initialize: function(id, scene, layer, data) {
        this.el = scene.createElement(data.width, data.height);
        this.el.setOriginPoint("middle");
        // Carré rouge pour le test
        this.el.fillStyle = "red";
        this.el.fillRect(0, 0, data.width, data.height);
        // --
        this.setPosition(data.x, data.y);
        this.on = canvas.Timeline.new(this.el).to({
            scaleX: 1.2,
            scaleY: 1.2
        }, 40, Ease.easeOutElastic); ①
        this.off = canvas.Timeline.new(this.el).to({
            scaleX: 1,
            scaleY: 1
        }, 40, Ease.easeOutElastic); ①
        layer.append(this.el);
    },
    setPosition: function(x, y) {
        this.el.x = x;
        this.el.y = y;
    },
    hit: function(val) { ②
        if (val) {
            this.on.call();
        }
        else {
            this.off.call();
        }
    }
});
```

La classe contient un peu plus de code. Dans le constructeur, deux animations pour grossir le texte ont été insérées ① (voir chapitre 4). Enfin, la méthode `hit` applique l'animation selon le type du contact ②.

Dans la méthode `addCoin` de la scène, nous faisons la liaison entre le modèle et le Sprite :

### Déclenchement des animations

```

addCoin: function(id, layer, data) {
    var coins = this.game_map.addEntity(id, data);
    var sprite = this.sprites[id] = Class.new("Sprite_Coins", [id, this, layer,
    ↵ data]);
    coins.callHit(function() {
        sprite.hit(true); ③
    }, function() {
        sprite.hit(false); ③
    });
}

```

Des fonctions sont attribuées à l'entité. Leur déclenchement appelle la méthode `hit` du Sprite pour afficher l'animation ③.

## Selon une action

L'interaction se déclenche cette fois-ci lorsqu'une touche est enfoncée par le joueur. Le principe est identique à celui de l'interaction selon le contact du héros, du moins pour le modèle. Voici la modification sur la scène :

### Modification des méthodes dans la scène

```

sprites: {},
coins: {},
addCoin: function(id, layer, data) {
    this.coins[id] = this.game_map.addEntity(id, data);
    this.sprites[id] = Class.new("Sprite_Coins", [id, this, layer, data]);
},
pressCoins: function() {
    var self = this;
    canvas.Input.press(Input.Enter, function() {
        for (var id in self.coins) {
            if (self.coins[id].isHit()) {
                self.sprites[id].hit(true);
            }
        }
    });
},
ready: function(stage) {
    // Code

    this.pressCoins();

    // Code
}

```

La méthode `ready` déclare un événement du clavier une seule fois. Dans ce cas, il faut parcourir les différentes entités pour vérifier si un contact est présent. Si le test est positif, nous appelons l'animation du Sprite grâce à l'identifiant.

# 8

## Mise en place des règles du jeu

---

Sans règle, le joueur ne saurait comprendre le sens du jeu ni arriver aux objectifs demandés. Voyons comment intégrer les règles du jeu dans le code.

Les règles du jeu servent à définir non seulement un but dans le parcours du joueur à travers les différentes cartes ou niveaux, mais aussi la difficulté pour y parvenir.

Il existe de multiples règles de jeu. D'ailleurs, ce sont les règles qui forment le concept. Nous allons distinguer 4 règles courantes :

- situation initiale du joueur;
- application du scénario;
- affichage du score;
- partie gagnée ou perdue.

## Situation initiale du joueur

On parle de l'initialisation des données du joueur. Nous l'avons déjà partiellement abordée dans les chapitres précédents (positions de départ, direction). Si nous reprenons le premier chapitre, nous remarquons que la description des types de jeu contient des paramètres spécifiques. Par exemple, le joueur dans un RPG possède des points d'expérience, des niveaux, une attaque, une défense, des équipements, un inventaire, etc.

Dans tous les cas, c'est le modèle qui gère le joueur.

### Le modèle avec les paramètres initiaux pour un RPG

```
Class.create("Game_Player", {
    speed: 3,
    x: x,
    y: y,
    dir: "right",
    level: 1,
    hp: 0,
    hp_max: 100,
    attack: [0, 622, 684, 746, 807, 869, 930, 992, 1053, 1115, 1176],
    exp: [0, 0, 25, 65, 127, 215, 337, 499, 709, 974, 1302],
    items: [],
    initialize: function(x, y) {
        this.x = x;
        this.y = y;
        this.hp = this.hp_max;
    },
    setLevel: function(level) {
        this.level = level;
    },
    getAttack: function(){
        return this.attack[this.level];
    }
});
```

Les paramètres `attack` et `exp` sont des tableaux dont chaque élément représente une valeur correspondant au niveau du joueur. Par exemple, si le joueur est au niveau 5, il sera possible de récupérer la valeur de l'attaque pour ce niveau :

### Affectation du niveau 5 au joueur

```
this.player = Class.new("Game_Player", [10, 10]);
this.player.setLevel(5);
console.log(this.player.getAttack());
```

Les positions (10;10) sont des données initiales importantes. Il faut définir en deux temps la carte ou le niveau à charger, puis le placement du joueur sur cette carte.

## Inventaire

Dans le code précédent, une propriété nommée `items` est un tableau, vide à l'initialisation, qui contiendra les objets récoltés par le joueur. Beaucoup de jeux ont une notion d'inventaire. Prenons quelques exemples :

- Point&Click : le joueur récupère des objets en les sélectionnant dans le décor. Les paramètres sont basiques car ils n'altèrent pas le héros. C'est seulement la possession de l'objet qui influencera le déroulement du scénario.

**Figure 8-1**  
Exemple d'un inventaire  
(Runaway)



- RPG : un objet possède généralement un prix et des paramètres changeant les capacités de base du héros.
- Plate-forme : les objets changent l'état du héros ou son score et permettent quelquefois d'accéder à un nouveau niveau.

### Définition de la classe Game\_Item

```
Class.create("Game_Item", {
    id: 0,
    used: 0,
    price: 0,
    description: "",
    name: "",
    hp: null,
    initialize: function(data) {
        this.id = data.id;
        this.price = data.price;
        this.description = data.description;
        this.name = data.name;
```

```

        this.hp = data.hp;
    },
    use: function() {
        this.used++;
    },
    getInfo: function() {
        return {
            description: this.description,
            name: this.name,
            price: this.price
        };
    },
    isUsed: function() {
        return this.used > 0;
    }
});
}
}

```

La classe `Game_Item` informe le joueur de la description et du nom de l'objet (pour l'afficher dans un menu) et précise si l'objet a déjà été utilisé ou non.

Le constructeur a un paramètre désignant les données de l'objet, lesquelles sont récupérées d'une base de données ou directement inscrites dans la scène.

Des propriétés comme `price` ajoutent une dimension financière virtuelle pour concevoir des magasins ou des moyens de débloquer ces objets dans le jeu selon l'argent virtuel récolté.

La propriété `hp` est un exemple d'une donnée pour l'objet. Ici, si sa valeur vaut 5, on peut admettre que l'objet va faire récupérer 5 points de vie au héros.

## Application du concept

### Affichage des points de vie

Ajoutons les méthodes dans la classe `Game_Player` pour que le joueur puisse ajouter des objets dans son inventaire et les utiliser.

D'après notre concept – qui revient souvent dans les jeux – le personnage gagne ou perd des points de vie.

#### Ajout de méthodes dans la classe `Game_Player`

```

addItem: function(data) { ①
    this.items.push(Game_Item(data));
},
getItem: function(id) {
    for (var i=0 ; i < this.items.length ; i++) { ②
        if (this.items[i].id == id) {

```

```

        return this.items[i];
    }
},
useItem: function(id) {
    var item = this.getItem(id);
    item.use();
    if (item.hp) this.gainHP(item.hp); ③
},
gainHP: function(num) { ④
    var hp = this.hp;
    hp += num;
    if (hp > this.hp_max) { ⑤
        hp = this.hp_max;
    }
    else if (hp < 0) {
        hp = 0;
    }
    this.hp = hp;
    return hp;
},
lostHP: function(num) {
    return this.gainHP(-num);
}
}

```

- 1 L'ajout d'objet se fait avec la méthode `addItem` ①, qui ajoute un élément du type `Game_Item` dans le tableau.
- 2 Nous récupérons l'objet en parcourant ce tableau grâce à son identifiant ②.
- 3 Si l'objet possède la propriété `hp` ③, nous appelons la méthode `gainHP` pour redonner des points de vie au héros ④.
- 4 Nous vérifions si les points de vie gagnés ne dépassent pas le maximum déclaré dans la propriété `hp_max` ⑤.

Pour donner une indication au joueur, nous devons afficher les points de vie du héros sur la scène.

#### Affichage d'une barre de vie sur la scène

```

canvas.Scene.new({
    name: "map",
    width_bar: 100, ④
    ready: function(stage) {
        var self = this;
        this.player = Class.new("Game_Player", [50, 50]); ①

        this.bar = this.createElement();
        this.bar.fillStyle = "green"; ③
        this.bar.fillRect(0, 0, this.width_bar, 20);
    }
})

```

```

        this.bar.strokeStyle = "black"; ②
        this.bar.strokeRect(0, 0, this.width_bar, 20);
        stage.append(this.bar);

        // Pour le test, le joueur perd 10 PV à l'appui de la touche
        Espace
        canvas.Input.keyDown(Input.Space, function() { ⑥
            self.refreshBar(self.player.lostHP(10), self.player.hp_max); ⑦
        });

    },
    refreshBar: function(hp, hp_max) {
        var width = hp * this.width_bar / hp_max; ⑤
        this.bar.fillRect(0, 0, width, 20);
    },
    render: function(stage) {
        stage.refresh();
    }
});

```

Après avoir créé une instance de la classe `Game_Player` ①, nous concevons un rectangle avec une bordure noire ② et un fond vert ③.

**Figure 8-2**

Représentation de  
la barre de vie



La taille du fond vert changera selon le nombre de points de vie du joueur. Pour le moment, nous décidons d'une taille fixe définie dans la propriété `width_bar` ④, qui sert à déterminer la nouvelle taille dans le rafraîchissement de la barre ⑤.

Ici, pour tester la perte de points de vie, nous avons simulé une diminution de la barre de vie lors de l'appui sur la touche `Espace` ⑥. Nous appelons la méthode `lostHP` ⑦ de la classe `Game_Player`, qui renvoie le nombre de points de vie possédés par le joueur.

#### À NOTER Méthode `lostHP`

La méthode `lostHP` sert à diminuer le nombre de points de vie. Nous l'avons définie pour faciliter la lecture du code car son nom est plus parlant que `gainHP(-10)`.

### Explication des règles au joueur

Dans tous les jeux, il faut expliquer les règles au joueur. C'est lorsque ce dernier introduit une nouveauté du Gameplay qu'elle lui est expliquée.

Cette démarche n'a rien de complexe. Nous affichons une boîte de dialogue en mettant le jeu en pause.

**Figure 8-3**  
Boîte de dialogue



#### Méthode d'ouverture d'une boîte de dialogue dans la scène

```
openDialog: function(stage, msg) {
    var _canvas = this.getCanvas(), ①
        self = this,
        dialog = this.createElement(),
        text = this.createElement();
    dialog.drawImage("box");
    dialog.x = _canvas.width / 2 ② - dialog.img.width / 2 ③;
    dialog.y = _canvas.height - dialog.img.height - 10 ④;

    text.font = 'normal 15px Arial';
    text.fillStyle = '#FFF';
    text.fillText(msg, 20, 30);
    dialog.append(text); ⑤
    stage.append(dialog);

    this.pause = true; ⑥
    stage.refresh();

    canvas.Input.KeyPress(Input.Enter, function() {
        self.pause = false; ⑦
        dialog.remove(); ⑧
    });
}
```

Cette méthode crée un élément juste le temps d'afficher la boîte de dialogue avec une image en fond. D'ailleurs, n'oubliez pas de charger cette image :

### Charger l'image de la boîte de dialogue

```
canvas.Scene.new({
    name: "MyScene",
    materials: {
        images: {
            "box": "images/System/MessageBack.png"
        }
    }
} //...
```

Pour centrer la boîte de dialogue et la placer en bas, nous prenons la largeur du `canvas` ① et nous centrons l'élément en le décalant de moitié vers la gauche ③ par rapport au milieu du `canvas` ②. Pour la hauteur, nous attribuons une marge de 10 pixels ④.

Un texte est affiché dans la boîte de dialogue ⑤ et nous mettons la propriété `pause` sur `true` ⑥ pour bloquer le rendu et donner la sensation d'une pause.

Quand le joueur appuie sur la touche `Entrée`, la pause est supprimée ⑦ et la boîte de dialogue effacée ⑧.

### Bloquer le rendu avec la propriété `pause`

```
render: function(stage) {
    if (this.pause) {
        return;
    }
    // Code
    stage.refresh();
}
```

Si la propriété `pause` est sur `true`, nous empêchons le code qui suit de s'exécuter en plaçant `return` dans la condition.

## Affichage du score

Le score est une indication de la performance du joueur. Dans les règles du jeu, on peut décider que le joueur doit atteindre un certain score pour être autorisé à poursuivre son aventure.

Dans un premier temps, initialisons dans méthode `ready` de la scène, un élément qui correspond au score :

### Initialisation de l'élément score

```
this.score = this.createElement();
this.score.font = 'bold 24px Arial';
this.score.fillStyle = '#000';
this.score.x = 10;
this.score.y = 10;
```

```
stage.append(this.score);
```

L'élément est un texte noir de 24 pixels. Il est affiché en haut à gauche de l'écran. Ajoutons une méthode pour rafraîchir le score :

#### Affichage du score

```
displayScore: function(number) {  
    this.score.fillText(number, 0, 100);  
}
```

Après la création de l'élément, initialisons le score :

#### Initialisation du score

```
this.displayScore(0)
```

La méthode `displayScore` est donc utilisée à chaque fois que l'on souhaite changer et afficher le nouveau score.

## Fin de partie : gagnée ou perdue

Il y a mille et une façons d'indiquer la victoire ou la défaite du joueur. Bien sûr, c'est lié au type du jeu. Nous allons voir le principe avec quelques exemples.

Dans la section précédente, nous avons vu comment afficher une barre de vie et un score. Dans la plupart des jeux, quand les points de vie tombent à 0, le joueur perd.

## Le joueur termine un parcours

Dans un jeu d'arcade, un personnage peut se déplacer de la gauche vers la droite, en évitant des obstacles, pour terminer un parcours.

**Figure 8-4**  
Le poisson termine  
un parcours.



Nous devons vérifier, dans la méthode `render` de la scène, la position du joueur par rapport à une valeur constante représentant l'extrémité du parcours :

#### Tester la fin d'un parcours dans la méthode render

```
var END_X = 5000;
var _canvas = this.getCanvas();

if (this.player.x >= END_X) { ①
    this.player.x += 10; ②
    if (this.player.x > _canvas.width) {
        canvas.Scene.call("World"); ③
        return;
    }
}
```

Ici, si la position X du joueur est supérieure à la constante (qui définit la position X de la fin du parcours) ①, nous faisons sortir le joueur de l'écran ② pour changer de scène ensuite ③.

### Le joueur sort de l'écran vers le bas

Dans quelques jeux de plates-formes, lorsque le personnage touche le bas de l'écran, le joueur perd et doit recommencer soit tout le niveau, soit à partir d'un endroit précis du niveau.

Le principe est identique à la finition d'un parcours. Nous testons les positions du joueur pour déterminer s'il sort de l'écran vers le bas. Plus précisément, c'est la position Y qui est concernée :

#### Tester si le joueur sort de l'écran vers le bas dans la méthode render

```
var _canvas = this.getCanvas();
if (this.player.y >= _canvas.height) {
    console.log("Touche le bas de l'écran");
}
```

# 9

## Ambiance du jeu

---

L'ambiance du jeu contribue beaucoup à la réussite de ce dernier auprès des joueurs ; pensez à ajouter des sons, des musiques, mais aussi des effets graphiques. Les effets sonores ne sont pas indispensables pour le fonctionnement du jeu, le bon déroulement du scénario et la fluidité d'un Gameplay. Pourtant, ils agrémentent considérablement l'ambiance du jeu en accentuant une action, un suspense, etc.

Les effets graphiques quant à eux évitent d'avoir des décors inchangés. Les effets de lumière, l'assombrissement, le brouillard ou un flash ajoutent du cachet dans l'ambiance du jeu.

## Ajouter des effets sonores

Deux solutions s'offrent à nous pour la manipulation de l'audio. Nous utiliserons :

- soit l'API de base de HTML5 ;
- soit l'API de SoundManager.

Avant tout, créons dans notre projet un dossier consacré au son et préchargeons-le au début de la scène.

### Précharger le son au début de la scène

```
canvas.Scene.new({
    name: "Map",
    materials: {
        sounds: {
            mymusic: "sounds/music.mp3",
        }
    },
    ready: function(stage) {
    }
});
```

## HTML5 Audio

Une des particularités de HTML5 est de proposer un élément de type [Audio](#). Trois formats peuvent être lus : MP3, OGG et WAV. Toutefois, des disparités existent quant à leur compatibilité sur les différents navigateurs.

### CONTRAINTE Prise en charge audio par les navigateurs

Il faudra passer par SoundManager pour gérer les formats en fonction des divers navigateurs.

**Tableau 9-1.** Compatibilité des sons sur les navigateurs

Navigateur	MP3	OGG	WAV
Internet Explorer 9+	Oui	Non	Non
Google Chrome	Oui	Oui	Oui
Firefox 4+	Non	Oui	Oui
Safari 5+	Oui	Non	Oui
Opera 10.6+	Non	Oui	Oui

Cette limitation contraint à :

- soit avoir le même son en format MP3 et OGG ;

- soit utiliser Flash si le format n'est pas reconnu;
- soit privilégier un navigateur (peu recommandé).

Pour insérer la musique, utilisons la méthode `get` sur l'objet `Sound` :

#### Insertion de la musique par son identifiant

```
var music = canvas.Sound.get("mymusic");
```

L'API de HTML5 Audio permet ensuite de jouer, stopper, etc.

#### Jouer la musique

```
music.play();
```

Voici les différentes méthodes et propriétés disponibles :

Tableau 9-2. Propriétés de HTML5 Audio

Propriété	Explication
<code>volume</code>	Changer le volume. Valeur comprise en 0 et 1
<code>currentTime</code>	Se positionner sur un temps (en secondes)
<code>buffered</code>	Récupère les temps initial et final du son. Exemple : <code>sound.buffered.start(0);</code> <code>sound.buffered.end(0);</code>
<code>paused</code>	Savoir si la lecture du son est en pause
<code>muted</code>	Savoir si le son est coupé
<code>duration</code>	Connaître la durée du son (en secondes)

Tableau 9-3. Méthodes de HTML5 Audio

Méthodes	Explication
<code>play()</code>	Joue le son
<code>pause()</code>	Met en pause le son
<code>canPlayType()</code>	Vérifie si le type du son est jouable. Exemple : <code>sound.canPlayType('audio/mpeg');</code>

## SoundManager

SoundManager a la particularité d'utiliser Flash si le navigateur ne reconnaît pas le format.

► <http://www.schillmania.com/projects/soundmanager2>

Pour l'utiliser, c'est très simple. Insérez tout d'abord le dossier `swf` de SoundManager à la racine de votre jeu. Si vous souhaitez changer le chemin, définissez-le dans linitialisation du `canvas` :

#### Changer le chemin du dossier swf

```
CE.defines("canvas", {
    'swf_sound: 'path/to/swf/'
});
```

Insérez le lien vers le Javascript :

#### Insertion de SoundManager

```
<script src="libs/soundmanager2.js"></script>
```

Pour récupérer la musique, utilisez la méthode `get` sur l'objet `Sound` :

```
var music = canvas.Sound.get("mymusic");
```

La démarche est identique à HTML5 Audio. Cependant les méthodes et propriétés utilisées sont celles de lAPI de SoundManager.

## Effectuer des fondus musicaux

Les fondus permettent d'enchaîner sur une nouvelle musique ou de prévenir qu'un nouvel événement va intervenir dans le jeu.

Le volume diminue ou augmente progressivement sur une durée prédéfinie, jusqu'à 0 ou une valeur prédéfinie.

#### Diminuer progressivement le volume de la musique

```
canvas.Sound.fadeOut("mymusic", 30); // Descend jusqu'à 0 sur une durée de
                                     30 frames
```

#### Augmenter progressivement le volume de la musique

```
canvas.Sound.fadeIn("mymusic", 30); // Augmente progressivement sur 30 frames
```

#### Atteindre progressivement un volume précis

```
canvas.Sound.fadeTo("mymusic", 30, 0.3);
```

Vous pouvez ajouter une fonction en paramètre supplémentaire à chaque fois. Elle s'exécutera lorsque le fondu sera terminé.

## Dynamiser avec des effets graphiques

### Ton de l'écran : effet jour/nuit

Créer un effet jour/nuit revient à changer le ton de l'écran :

- nuit : assombrir l'écran;
- jour : éclairer l'écran.

L'idée est de mettre un élément qui se superpose sur tout le décor avec une opacité plus faible.

#### Jour

L'élément transparent est blanc et a une forte luminosité.

##### Effet Jour

```
var _canvas = this.getCanvas(),
    effect = this.createElement();
effect.fillStyle = "white";
effect.globalCompositeOperation = "lighter";
effect.opacity = .5;
effect.fillRect(0, 0, _canvas.width, _canvas.height);
stage.append(effect);
```

#### Nuit

L'élément transparent est noir et a une faible luminosité.

##### Effet Nuit

```
var _canvas = this.getCanvas(),
    effect = this.createElement();
effect.fillStyle = "black";
effect.globalCompositeOperation = "darker";
effect.opacity = .5;
effect.fillRect(0, 0, _canvas.width, _canvas.height);
stage.append(effect);
```

### Flash visuel

L'idée est de créer temporairement un élément dont l'opacité diminue jusqu'à faire disparaître complètement l'élément.

Le code suivant peut être placé dans une méthode ou une fonction qui est exécutée à un moment précis du jeu (lorsque le joueur se fait toucher par un adversaire, par exemple).

### Créer un flash visuel

```
var _canvas = self.getCanvas(),
    effect = self.createElement();
effect.fillStyle = "red";
effect.globalCompositeOperation = "lighter";
effect.opacity = .5;
effect.fillRect(0, 0, _canvas.width, _canvas.height);
stage.append(effect);

canvas.Timeline.new(effect)
.to({opacity: "0"}, 10)
.call(function() {
    this.remove();
});
});
```

Comme pour l'effet jour/nuit, nous créons un élément qui s'étend sur tout l'écran avec une opacité de 0,5. Nous appliquons un changement de l'opacité sur une ligne temporelle et très rapidement pour donner la sensation d'un flash. Quand l'opacité est arrivée à 0, nous supprimons l'élément.

# 10

## Les adversaires

---

Les adversaires donnent un défi et une difficulté supplémentaires au joueur pour atteindre son but. Comment rendre les adversaires vivants ?

Un jeu comporte presque toujours des adversaires. Ces derniers ajoutent de la difficulté et améliorent le Gameplay. Ils se caractérisent par des paramètres de comportement et par une intelligence artificielle plus ou moins poussée.

## Paramètres des adversaires

Les adversaires sont caractérisés par des paramètres indispensables pour calculer les dégâts lors des combats et savoir s'ils ont été éliminés ou non :

- attaque;
- défense;
- force;
- points de vie.

## Modèle et affichage des points de vie

Dans les chapitres précédents, nous avons vu comment utiliser les règles du jeu pour les appliquer sur un personnage. Le modèle des adversaires reprend cette idée en incorporant une barre de points de vie, qui diminue quand le joueur touche son opposant.

### Modèle

#### Modèle des adversaires

```
Class.create("Game_Ennemy", {
    attack: 0,
    defense: 0,
    strength: 0,
    hp: 0,
    hp_max: 0,
    setParams: function(params) {
        this.attack = params.attack;
        this.defense = params.defense;
        this.strength = params.strength;
        this.hp = params.hp_max;
        this.hp_max = params.hp_max;
        return this;
    }
}).extend("Game_Entity");
```

La classe `Game_Ennemy` hérite de `Game_Entity` qui gère les interactions, mouvements et positions de l'élément (voir chapitre 7). Cinq propriétés vont nous servir pour les calcul des dégâts et l'affichage des points de vie.

Tableau 10-1. Propriétés de l'adversaire

Propriété	Explication
attack	Points d'attaque
defense	Points de défense
strength	Points de force
hp	Nombre courant de points de vie
hp_max	Nombre total de points de vie

Lors de l'initialisation de l'entité, nous lui ajouterons les paramètres grâce à la méthode `setParams` précédemment conçue.

## Sprite

La classe `Sprite_Ennemis` récupère des données envoyées de la scène :

- `x` : position X;
- `y` : position Y;
- `hp_max` : points de vie.

### Sprite représentant les adversaires avec une barre de vie

```
Class.create("Sprite_Ennemis", {
    el: null,
    scene: null,
    hp: 0,
    hp_max: 0,
    width_bar: 30,
    initialize: function(id, scene, layer, data) {
        this.scene = scene;
        this.el = scene.createElement(data.width, data.height);

        // Simple image
        this.el.drawImage("en_right", 0, 0, 32, 32, 0, 0, 32, 32);

        this.setPosition(data.x, data.y);
        this.hp = this.hp_max = data.hp_max;
        this._displayBar();
        layer.append(this.el);
    },
    _displayBar: function() {
        this.bar = this.scene.createElement();

        this.bar.fillStyle = "green";
        this.bar.fillRect(2, -5, this.width_bar, 5);

        this.bar.strokeStyle = "black";
```

```

        this.bar.strokeRect(2, -5, this.width_bar, 5);

        this.el.append(this.bar);
    },
    setPosition: function(x, y) {
        this.el.x = x;
        this.el.y = y;
    }
});

```

Un élément est créé représentant l'adversaire et positionné aux valeurs X et Y sur la couche des objets. La méthode `_displayBar` initialise la barre de vie de couleur verte avec une bordure noire, et affichée au-dessus de l'élément.

## Scène

La scène contient deux objets stockant les Sprites et les modèles des entités.

### Scène qui ajoute les adversaires

```

canvas.Scene.new({
    name: "map",
    sprites: {},
    ennemis: {},
    addEnnemy: function(id, layer, data) {
        this.ennemis[id] = this.game_map.addEntity(id, data);
        setParams(data);
        this.sprites[id] = Class.new("Sprite_Ennemis", [id, this,
        layer, data]);
    },
    ready: function(stage) {
    },
    render: function(stage) {
    }
});

```

Ajouter un ennemi revient à appeler la classe `Sprite_Ennemis` avec les données. Rappelons que la méthode `addEntity` ajoute un ennemi sur la carte.

### Ajout d'un adversaire sur la carte

```

Class.create("Game_Map", {
    entities: [],
    map: null,
    initialize: function(map) {
        this.map = map;
    },
    addEntity: function(id, data) {
        var entity = Class.new("Game_Ennemy", [id, data.x, data.y, data.
    }
});

```

```

    ↵ width, data.height]);
    this.entities.push(entity);
    return entity;
}
});

```

Dans la méthode `ready`, nous ajoutons un ennemi en spécifiant ses positions et sa taille pour les interactions ainsi que les paramètres pour les calculs des dégâts.

#### Ajout d'un adversaire avec des paramètres précis

```

this.addEnnemy(1, layer_event, {
    x: 50,
    y: 250,
    width: 32,
    height: 32,
    attack: 10,
    defense: 5,
    strength: 10,
    hp_max: 100
});

```

## Calcul des dégâts

Le calcul des dégâts se fait logiquement dans le modèle, dans la classe `Game_Enemy`. Nous affichons le montant des dommages sur le Sprite et la barre de vie diminue. Lorsque l'adversaire n'a plus de points de vie, nous le supprimons de la carte.

Ajoutons dans la classe `Game_Enemy` les deux méthodes suivantes :

#### Calcul des dégâts

```

damage: function(player) {
    var power = player.attack - (this.defense / 2), ①
        variance = 15; ②

    var damage = player.strength + (variance + Math.floor(Math.random() *
    ↵ Math.round(variance/3)) * (Math.random() > .5 ? -1 : 1)); ③

    this.changeHP(-damage); ④
    return damage;
},
changeHP: function(num) {
    var hp = this.hp;
    hp += num;
    if (hp > this.hp_max) {
        hp = this.hp_max;
    }
    else if (hp < 0) {
        hp = 0;
    }
}

```

```

        }
        this.hp = hp;
    }
}

```

#### SIMPLIFICATION Math.min et Math.max

Ce genre de code pourrait être simplifié en utilisant les méthodes mal-aimées de Javascript : `Math.min` et `Math.max`, par exemple :

```

this.hp = num > 0 ? Math.min(this.hp + num, this.hp_max) : Math.max(this.hp + num,
    0);

```

La méthode `damage` prend en compte les propriétés `attack` et `strength` du joueur (`Game_Player`) pour les utiliser dans les calculs.

La puissance de l'attaque est la soustraction des points d'attaque du joueur à la moitié des points de défense de l'adversaire ①. Remarquez qu'il n'y a pas vraiment de règles sur les calcul de dégâts ; c'est plus lié à la cohérence des valeurs. La variable `variance` ajoute une valeur aléatoire mais proche de la valeur initiale ②. Après le calcul, nous changeons la propriété `hp` ③ et renvoyons à la scène le montant des dommages.

#### Modification de la barre de vie

Dans la méthode d'interaction du Sprite (voir chapitre 7), nous rafraîchissons la barre de vie selon le montant calculé des dommages.

#### Rafraîchissement de la barre de vie selon les dégâts

```

hit: function(damage) {
    this.hp -= damage;
    if (this.hp < 0) {
        // mettre une animation
        this.el.remove();
        return true;
    }
    var width = this.width_bar * this.hp / this.hp_max;
    this.bar.fillRect(2, -5, width, 5);
    return false;
}

```

La barre rétrécit selon le rapport entre les points de vie courants et les points de vie initiaux. Un booléen est renvoyé pour signaler si les points de vie ont atteint 0. Si c'est le cas, nous supprimons l'élément et renvoyons la valeur `true`.

#### Enlever des points de vie par l'appui sur une touche

Dans la scène, nous allons enlever des points de vie à l'adversaire quand la touche `Entrée` est enfoncee. Le morceau de code qui suit reprend l'interaction du chapitre 7.

### Attaquer l'adversaire par l'appui sur une touche

```

pressEnnemis: function(stage) {
    var self = this, damage;
    canvas.Input.press(Input.Enter, function() {
        for (var id in self.ennemis) {
            if (self.ennemis[id].isHit()) {
                damage = self.ennemis[id].damage(self.game_player); ❶
                self.displayDamage(damage, self.sprites[id].el); ❷
                if (self.sprites[id].hit(damage)) { ❸
                    self.deleteEnnemy(id); ❹
                }
            }
        });
    },
    deleteEnnemy: function(id) {
        this.game_map.removeEntity(id); ❺
        delete this.ennemis[id];
        delete this.sprites[id];
    }
}

```

L'objet adversaire est parcouru de façon à vérifier une collision.

- ❶ Si c'est le cas, la méthode `damage` ❶ de la classe `Game_Ennemy` est exécutée pour connaître le montant des dégâts à enlever à l'adversaire.
- ❷ Nous l'affichons ensuite avec une animation ❷.
- ❸ Nous diminuons la barre de vie du Sprite ❸. Si l'ennemi est mort, nous l'enlevons de la carte. Pour cela, la méthode `deleteEnnemy` ❹ supprime les objets sur la scène mais aussi l'entité sur la carte. C'est la classe `Game_Map` qui se charge de cette dernière action avec une méthode nommée `removeEntity` ❺.

#### À NOTER Condition de suppression de l'adversaire

L'ennemi n'est pas systématiquement supprimé de la carte car, lors de son initialisation, ses points de vie ne sont pas à 0 ; la condition de suppression n'est donc pas remplie.

### Suppression de l'adversaire sur la carte

```

removeEntity: function(id) {
    for (var i=0 ; i < this.entities.length ; i++) {
        if (this.entities[i].id = id) {
            this.entities.splice(i, 1);
            return true;
        }
    }
}

```

N'oubliez pas d'initialiser l'appui sur la touche en ajoutant l'appel à `pressEnnemis` dans la méthode `ready` de la scène.

#### Initialisation de l'appui sur la touche

```
this.pressEnnemis(stage);
```

#### Affichage des dommages

Lorsque nous enlevons des points de vie à l'adversaire, il est utile de le montrer au joueur. Dans les codes précédents, nous avons appelé la méthode `displayDamage` sur le Sprite. Ajoutez-la dans la scène :

#### Affichage des animations des dégâts

```
displayDamage: function(text_damage, sprite) {
    var text = this.createElement();
    text.font = 'bold 14px Arial';
    text.fillStyle = '#FFF';
    text.textBaseline = 'middle';
    text.fillText(text_damage, 0, 0);
    text.strokeStyle = '#000';
    text.strokeText(text_damage, 0, 0);

    sprite.append(text);

    canvas.Timeline.new(text)
        .add({y: -30, opacity: -1}, 40, Ease.easeOutQuint)
        .call(function() {
            this.remove();
        });
}
```

Un élément est créé temporairement. C'est un simple texte blanc avec un contour noir. Il est rattaché au Sprite et une animation lui est appliquée. Il défile de 30 pixels vers le haut avec une accélération. Quand l'animation est terminée, nous supprimons le texte.

#### Zones spécifiques d'interaction

Un adversaire peut avoir des vulnérabilités ou des défenses plus accentuées sur une partie de son corps. C'est dans ce cas-là que les zones spécifiques d'interaction interviennent. Par exemple, si le joueur attaque sur le devant de l'adversaire, les dégâts seront plus faibles que s'il arrive par derrière.

Pour y parvenir, nous allons tester les interactions du joueur selon des zones spécifiques de l'adversaire après la collision.

**Figure 10-1**

Deux zones spécifiques d'interaction



La classe Game\_Ennemy prend deux propriétés supplémentaires.

#### Propriétés pour définir les zones d'interaction

```
_hit_key: "",  
_hit_area: {  
    left: {  
        behind: {start: [16,0], size: [16, 32]},  
        front: {start: [0,0], size: [16, 32]}  
    },  
    right: {  
        behind: {start: [0,0], size: [16, 32]},  
        front: {start: [16,0], size: [16, 32]}  
    }  
}
```

**Tableau 10-2. Propriétés des zones interactives**

Propriété	Explication
_hit_key	Indique si l'interaction est devant (front) ou derrière l'adversaire (behind).
_hit_area	Configuration des zones selon la direction : start désigne les positions du point supérieur gauche du rectangle des collisions et size est la taille du rectangle

Nous devons tester si un point est présent dans la zone d'interaction.

#### Test d'un point dans la zone d'interaction

```
hitArea: function(x, y) {  
    var self_x, self_y, points;  
    for (var key in this._hit_area[this.dir]) {  
        points = this._hit_area[this.dir][key];  
        self_x = this.x + points.start[0];  
        self_y = this.y + points.start[1];  
        if (x >= self_x && x <= self_x + points.size[0] &&  
            y >= self_y && y <= self_y + points.size[1]) {  
            this._hit_key = key;  
            return key;  
        }  
    }  
}
```

```

        }
    },
    getHitKey: function() {
        return this._hit_key;
    }
}

```

Ces méthodes sont à insérer dans `Game_Ennemy`. Nous parcourons les zones selon la direction (propriété `dir`). Pour chaque point, nous testons sa présence dans la zone. Si c'est le cas, la propriété `_hit_key` récupère le nom de la zone.

#### Variation des dégâts selon la zone d'interaction

```

damage: function(player) {
    var power = player.attack - (this.defense / 2),
        variance = 15;

    var damage = player.strength + (variance + Math.floor(Math.random() *
    ↪ Math.round(variance/3)) * (Math.random() > .5 ? -1 : 1));
    if (this._hit_key == "front") {
        damage = Math.round(damage / 5);
    }
    this.changeHP(-damage);
    return damage;
}

```

La méthode `damage` est modifiée pour vérifier la zone. Si l'attaque se fait en face de l'adversaire, les dommages sont divisés par 5.

En reprenant les interactions du chapitre 7, nous avons une fonction privée dans la méthode `isPassable` de la classe `Game_Map` :

#### Appel des zones d'interaction lors d'une collision avec l'adversaire

```

function pointIsPassable(ent, x, y) {
    if (x >= ent.x && x <= ent.x + ent.width &&
        y >= ent.y && y <= ent.y + ent.height) {
        ent.hitArea(x, y);
        ent.hit(true);
        return false;
    }
    ent.hit(false);
    return true;
}

```

Nous exécutons la méthode `hitArea` pour les zones spécifiques quand une collision se produit.

## Champ de vision

Le champ de vision permet à l'adversaire de détecter le personnage principal et de réagir en conséquence. Les ajouts se font dans la classe `Game_Ennemy`.

### Zone de détection

La zone de détection est définie en pixels autour de l'adversaire. Si le personnage principal entre dans cette zone, l'adversaire réagit.

**Figure 10-2**  
La zone de détection  
de l'adversaire



Lors de la détection, l'adversaire a un statut :

- `detect` : a détecté;
- `notdetect` : ne détecte plus;
- `reaction` : a déjà détecté et réagi.

Tout d'abord, nous déclarons une propriété `detection_area` dans la classe.

#### Propriété pour la détection

```
detection_area: {
    size: TILE_W * 5,
    state: ""
}
```

La propriété est un objet définissant la taille de la zone – la constante `TILE_W` étant la taille d'un carreau indiquée plus haut dans le code – et l'état de la détection, `state`.

Ajoutons deux méthodes utiles pour changer et tester l'état.

#### Méthodes pour changer et tester l'état

```
changeDetectionState: function(name) {
    this.detection_area.state = name;
},
isDetectionState: function(name) {
    return this.detection_area.state == name;
}
```

Dans une nouvelle méthode, nous testons les positions du joueur par rapport à la zone de l'adversaire et nous changeons l'état en conséquence.

#### Méthode de la détection

```
detection: function(player) {
    var area = this.detection_area.size;
    if (player.x >= this.x - area && player.x <= this.x + area &&
        player.y >= this.y - area && player.y <= this.y + area) {
        if (!this.isDetectionState("reaction")) { ①
            this.changeDetectionState("detect");
        }
        return true;
    }
    else {
        this.changeDetectionState("notdetect");
        this.state = "";
    }
    return false;
}
```

Pour que l'ennemi ne détecte qu'une fois le joueur qui rentre dans la zone, nous changeons l'état à `detect` seulement si le joueur est en dehors de la zone et si l'adversaire n'est pas déjà en train de réagir ①.

## Réaction

Quand l'adversaire détecte le héros, nous changeons l'état de détection à `reaction` pour effectuer une réaction bien précise.

#### Méthode de la réaction

```
reaction: function(player, callbacks) {
    var dir = "";
    if (this.detection(player) && this.isDetectionState("detect")) {
        this.changeDetectionState("reaction");
        if (player.hp * 100 / player.hp_max < 10) { ①
            this.state = "mid-aggressive";
        }
        else if (player.attack <= this.attack) { ②
            this.state = "aggressive";
        }
    }
    if (player.x < this.x) dir = "left";
    if (player.x > this.x) dir = "right";

    if (callbacks[this.state]) callbacks[this.state].call(this, dir);
}
```

Cette méthode sera appelée en boucle dans le rendu de la classe. Le point intéressant est de personnaliser la réaction selon des paramètres externes. Dans notre exemple, nous testons deux circonstances :

- si les points de vie du joueur sont en-dessous de 10 % de ses points de vie maximaux ①;
- si les points d'attaque de l'adversaire sont plus élevés que ceux du joueur ②.

Selon le cas, l'état de l'adversaire est différent. Nous appelons une fonction de rappel selon cet état en indiquant la direction de l'adversaire.

Dans la méthode `render`, nous parcourons les adversaires pour appeler la méthode `reaction`.

#### Déclenchement de la réaction de l'adversaire

```
for (var id in this.ennemis) {  
    this.ennemis[id].reaction(this.game_player, {  
        "aggressive": function(dir) {  
  
        },  
        "mid-aggressive": function(dir) {  
  
        }  
    });  
}
```

Dans les fonctions déclenchées, vous pouvez afficher des animations, un déplacement (en vous aidant du paramètre `dir`), etc.



# 11

## Réaliser la sauvegarde

---

Pour que le joueur reprenne sa partie plus tard, la sauvegarde des données, d'une importance capitale, doit être proposée dans votre jeu.

Chaque jeu intègre une sauvegarde. C'est un point fondamental pour que le joueur reprenne sa dernière partie. La procédure de la sauvegarde en Javascript consiste à sérialiser les données dans un certain format et à les stocker dans un fichier.

## Sérialisation des classes

La sérialisation consiste à obtenir une chaîne de caractères représentative de l'objet. CanvasEngine propose une classe nommée `Marshal` pour effectuer cette procédure. Seules les propriétés qui ne sont pas des fonctions peuvent être sérialisées.

Imaginons que vous ayez la classe `Game_Player` suivante :

### Classe simple de la gestion du joueur

```
var Game_Player = function(x, y) {
    return Class.create("Game_Player", {
        speed: 3,
        x: x,
        y: y,
        dir: "right"
    });
};
```

Nous utilisons la méthode `dump` pour sauvegarder les propriétés de la classe. Elle a deux paramètres : la classe à sérialiser et le nom du fichier.

### Sauvegarde à l'appui de la touche Entrée

```
var self = this;
this.player = Game_Player(50, 50);
canvas.Input.press(Input.Enter, function() {
    Marshal.dump(self.player, "filename");
    Marshal.dump(self.test, "filename");
});
```

Le joueur est placé aux positions (50, 50) sur la scène. Pour tester la sauvegarde, nous demandons de stocker les valeurs lorsque la touche `Entrée` est enfoncee.

Pour un jeu monojoueur, les données sont stockées dans le navigateur. Le nom est donc un identifiant pour récupérer les données lors du chargement.

#### MULTIJOUEUR Sauvegarde pour le multijoueur

Dans le cas d'un jeu multijoueur, les données ne doivent pas être enregistrées en local. Soit vous utilisez un système de fichier personnalisé côté serveur, soit une base de données. Le chapitre 13 abordera la conception d'une sauvegarde pour un jeu multijoueur.

La classe `Marshal` fonctionne comme une pile : les premières données enregistrées seront les premières à être chargées avec la méthode `load`. L'ordre est donc très important. `Marshal` réalise une sérialisation des données d'une classe dans `localStorage` et la transition d'un système clé/valeur vers une classe lors du chargement.

### MARSHAL Comme en Ruby

Ruby a un module nommé `Marshal` pour convertir des collections d'objets Ruby dans un flux d'octets, ce qui permet de stocker en dehors du script et sauvegarder les données. CanvasEngine reprend ce concept, mais en Javascript.

### SANS FRAMEWORK Sauvegarder

L'idée est d'enregistrer des données au format JSON en local. Pour cela, nous récupérons les propriétés d'une classe pour les convertir en JSON :

```
function save(_class, file) {
    var new_data = {};

    for (var method in _class) {
        if (typeof _class[method] != "function") {
            new_data[method] = _class[method];
        }
    }

    if (localStorage) {
        localStorage[file] = JSON.stringify(new_data);
    }
}
```

La variable `new_data` est une copie des propriétés de la classe. Elle ignore évidemment les méthodes grâce à `typeof`. L'objet est ensuite sauvegardé en local avec `localStorage`. Essayons avec une classe de test :

```
function Test() {}

Test.prototype = {
    foo: "bar"
}

var test = new Test();
save(test, "filename");
```

La propriété `foo` est stockée avec le format suivant : `{"foo": "bar"}`

## Chargement des données

Le chargement s'effectue avec la méthode `load`. Chaque objet est recréé dans le même ordre que pour l'écriture du fichier.

### Chargement des données du joueur

```
this.player = Marshal.load("filename");
```

```
| this.test = Marshal.load("filename");
```

Toutes les propriétés vont être réattribuées au joueur. Ensuite, vous pouvez charger le niveau et les éléments correspondant aux données chargées et non aux données initialisées.

Connaître l'existence des données (méthode `exist`) offre le choix au joueur de commencer une nouvelle partie ou de charger sa dernière partie. Dans certains jeux, elles sont directement chargées si la vérification de leur présence est positive.

#### Les données existent-elles ?

```
| if (Marshal.exist("filename")) {
    // Code
}
```

#### SANS FRAMEWORK Charger des données

Le chargement consiste à attribuer les données à la classe :

```
function load(_class, file) {
    var data = {};
    if (localStorage && localStorage[file]) {
        data = JSON.parse(localStorage[file]);
    }

    for (var method in _class) {
        _class[method] = data[method];
    }

    return _class;
}
```

Nous vérifions d'abord si des données sont bien stockées en local. Si c'est le cas, nous transformons le format JSON en un objet Javascript.

Nous parcourons la classe et affectons la valeur de la donnée équivalente à la propriété de la classe.

```
function Test() {

}

Test.prototype = {
    foo: "bar"
}

var test = new Test();
load(test, "filename");
```

Ainsi, si nous admettons qu'une sauvegarde est déjà présente, le chargement va changer la propriété `foo` de la classe `Test`.

# 12

## Cas pratique : créer un jeu plate-forme

---

Illustrons la théorie avec un cas pratique.

Comment créer rapidement un jeu de plate-forme ?

Récapitulons les chapitres précédents avec un cas pratique. Nous allons construire un jeu de plate-forme en HTML5 avec le Level Design, les animations, les mouvements, les collisions, etc.

## Règles du jeu

En reprenant les règles du jeu de type plate-forme, nous considérons plusieurs fondamentaux :

- Le personnage principal bouge vers la gauche ou vers la droite grâce au contrôle du clavier.
- Le personnage principal peut sauter à l'aide d'une touche du clavier.
- Le personnage principal « tombe » vers le bas tant qu'il n'entre pas en collision avec une plate-forme.
- Si le personnage touche le bas de l'écran, le joueur perd.

## Initialisation et création des classes

Comme à l'accoutumée, nous initialisons le `canvas` en définissant les extensions utiles à notre jeu. L'ordre n'a pas d'importance. Le framework recopie les méthodes pour le `canvas` défini.

### Initialisation du canvas

```
var canvas = CE.defines(["canvas_id"]).
    extend(Input).
    extend(Animation).
    extend(Tiled).
    extend(Scrolling).
    ready(function(ctx) {
        canvas.Scene.call("Map");
   });
```

La classe `Game_Character` s'occupera des mouvements, de la gravité et du saut des personnages présents sur la carte.

### Classe pour la gestion des personnages

```
Class.create("Game_Character", {
    map: null,
    currentState: "",
    a: 0,
    speed: 4,
    x: 0,
    y: 0,
    dir: "right",
    initialize: function(id, width, height, x, y, map) {
        this.x = x;
        this.y = y;
        this.id = id;
        this.width = width;
        this.height = height;
```

```

        this.map = map;
    }
});
```

La classe Game\_Player gère uniquement les propriétés liées au joueur. Elle hérite de Game\_Character.

#### Gestion du joueur

```

Class.create("Game_Player", {
}) .extend("Game_Character");
```

La classe Game\_Map contient les méthodes concernant la carte.

#### Gestion des cartes

```

Class.create("Game_Map", {
});
```

Enfin, la scène charge les images et les sons du jeu avant d'appeler la méthode ready.

**Figure 12-1**  
Image utilisée  
pour le déplacement  
vers la droite



#### Scène chargeant les images

```

canvas.Scene.new({
    name: "Map",
    materials: {
        images: {
            area01_level_tiles: {path: "area01_level_tiles.png",
                transparentcolor: "#ff00ff"},

            gripe_run_left: {path: "gripe.run_left.png",
                transparentcolor: "#ff00ff"},

            gripe_run_right: {path: "gripe.run_right.png",
                transparentcolor: "#ff00ff"},

            gripe_stand_left: {path: "gripe.stand_left.png",
                transparentcolor: "#ff00ff"},

            gripe_stand_right: {path: "gripe.stand_right.png",
                transparentcolor: "#ff00ff"},

            en_right: {path: "en_right.png", transparentcolor: "#ff00ff"},

            en_left: {path: "en_left.png", transparentcolor: "#ff00ff"},

            Heal5: "Heal5.png",

            background: "area01_bkg0.png"
        },
    }
},
```

```

        sounds: {
            jump: "sounds/jump.wav"
        }
    },
    ready: function(stage) {

},
    render: function(stage) {

}
));

```

## Chargement du niveau

La méthode `ready` initialise les modèles précédemment créés. Nous utilisons Tiled Map Editor pour créer le niveau. Aux carreaux, nous donnons la propriété `passable` avec une valeur hexadécimale pour la praticabilité. Les données du niveau se trouvent dans un fichier JSON, dans un dossier nommé `map`.

### Chargement du niveau et initialisation des modèles

```

var self = this,
    tiled = canvas.Tiled.new();

tiled.load(this, stage, "map/map.json");

tiled.ready(function() {

    var background = self.createElement();
    background.drawImage("background");
    stage.append(background);
    self.game_map = Class.new("Game_Map", [this]); ①

    var tile_w = this.getTileWidth(), ②
        tile_h = this.getTileHeight(); ②
    self.player = self.createElement(32, 32); ③
    self.game_player = Class.new("Game_Player", ["player", 32, 32, 10 * tile_w,
10 * tile_h, self.game_map]); ④
    self.player.drawImage("gripe_stand_right");
    self.player.x = self.game_player.x;
    self.player.y = self.game_player.y;

    var layer_event = this.getLayerObject(); ⑤
    layer_event.append(self.player); ⑥

});

```

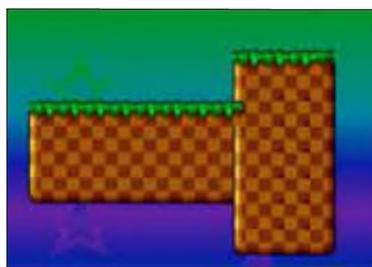
Lorsque le niveau est prêt, nous initialisons la classe `Game_Map` en lui attribuant l'instance `tiled` (avec la référence `this`) ①

Connaître la taille des carreaux ② nous permet de placer aisément le joueur sur la carte. Nous créons un élément pour afficher le personnage principal sur la carte ③ et initialisons le modèle `Game_Player` avec un identifiant `player` ④, la taille ( $32 \times 32$  px) et l'attribution de l'instance `game_map`.

La méthode `getLayer0Object` nous fournit la couche spécifique aux objets ⑤ et dans laquelle nous ajoutons l'élément représentant le joueur ⑥.

**Figure 12-2**

Exemple du niveau créé dans Tiled Map Editor



## Création des animations

Quand le personnage bouge vers la droite ou la gauche, nous devons l'animer pour montrer au joueur qu'il se déplace. Le code qui suit s'intègre après l'ajout de l'élément dans la couche d'objets dans la méthode `ready` de l'instance `tiled`.

### Création des animations

```
//Animation quand le personnage bouge vers la gauche
var anim_left = canvas.Animation.new({
    images: "gripe_run_left",
    animations: {
        left: {
            frames: [0, 6],
            size: {
                width: 32,
                height: 32
            },
            frequence: 4
        }
    }
});

//Animation quand le personnage bouge vers la droite
var anim_right = canvas.Animation.new({
    images: "gripe_run_right",
```

```

        animations: {
            right: {
                frames: [0, 6],
                size: {
                    width: 32,
                    height: 32
                },
                fréquence: 4
            }
        }
    });

// Ajout des animations sur l'élément Joueur
anim_left.add(self.player);
anim_right.add(self.player);

// Jouer l'animation quand la touche Gauche est pressée
canvas.Input.press(Input.Left, function() {
    anim_left.play("left", true);
});

// Jouer l'animation quand la touche Droite est pressée
canvas.Input.press(Input.Right, function() {
    anim_right.play("right", true);
});

```

Deux animations sont créées pour les deux directions (gauche et droite) avec deux images différentes. Dans les deux cas, nous prenons 6 séquences pour l'animation de  $32 \times 32$  px avec une fréquence de 4 (pour éviter des animations trop rapides).

Nous affectons les animations à l'élément `player`. Nous jouons la bonne animation selon la touche pressée.

## Défilement de la carte

À présent, nous devons faire défiler le décor et centrer l'écran sur le personnage principal lors du déplacement. Toujours à la suite du code plus haut :

### Définition du défilement sur la carte

```

self.scrolling = canvas.Scrolling.new(self ①, tile_w ②, tile_h ②);
self.scrolling.setMainElement(self.player); ③
var map = self.scrolling.addScroll({
    element: this.getMap(), ④
    speed: ④,
    block: true,
    width: this.getWidthPixel(),
    height: this.getHeightPixel()
});

```

```
self.scrolling.setScreen(map); ⑤
```

Nous appelons la classe `Scrolling` en désignant la scène ① et les dimensions ② d'un carreau. L'élément principal est l'instance `player` ③. L'écran se centrera par rapport à ce dernier.

Enfin, nous définissons le déplacement sur une carte récupérée à l'instance `tiled` ④ et indiquons la taille de la carte en pixels avec `getWidthPixel` pour la largeur (contrairement à `getWidth` pour la taille en carreaux) et `getHeightPixel` pour la hauteur.

Nous centrons la carte ⑤ sur le personnage principal dès le commencement du niveau.

Nous terminons le défilement en mettant à jour le `Scrolling` dans la méthode `render` de la scène :

#### Rendu pour le Scrolling

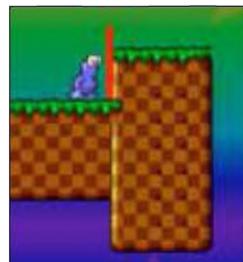
```
render: function(stage) {  
    if (!this.game_player) {  
        return;  
    }  
  
    this.scrolling.update();  
    stage.refresh();  
}
```

Remarquez que nous ignorons le rendu si l'instance `game_player` n'existe pas.

## Gestion des collisions

Le modèle `Game_Map` vérifie les positions du personnage principal et regarde la praticabilité d'un carreau pour les collisions.

**Figure 12-3**  
Collision sur le  
décor de droite



### Praticabilité

```

Class.create("Game_Map", {
    map: null,
    initialize: function(map) {
        this.map = map;
    },
    isPassable: function(player, new_x, new_y) {
        var ent, self = this;

        if (new_x < 0 || (new_x + player.width) > this.map.getWidthPixel()) {
            return false;
        }

        var tile_w = this.map.getTileWidth(),
            tile_h = this.map.getTileHeight();

        function pointIsPassableInTile(x, y) {
            var map_x = Math.floor(x / tile_w),
                map_y = Math.floor(y / tile_h);

            var dir_hexa = 0x0;
            switch (player.dir) {
                case "left":
                    dir_hexa = 0x8;
                    break
                case "right":
                    dir_hexa = 0x2;
                    break
                case "bottom":
                    dir_hexa = 0x4;
                    break
                case "up":
                    dir_hexa = 0x6;
                    break
            }

            var props = self.map.getTileProperties(null, map_x, map_y);
            for (var i=0 ; i < props.length ; i++) {
                if (props[i] && (parseInt(props[i].passable, 16) & dir_hexa)
                    != dir_hexa) {
                    return false;
                }
            }
            return true;
        }

        if (!pointIsPassableInTile(new_x, new_y) ||
            !pointIsPassableInTile(new_x + player.width, new_y) ||
            !pointIsPassableInTile(new_x, new_y + player.height) ||
            !pointIsPassableInTile(new_x + player.width, new_y + player.height))
            return false;
    }
});

```

```

        !pointIsPassableInTile(new_x + player.width, new_y + player.
    ↪ height)) {
    return false;
}

return true;
});

```

La méthode `isPassable` vérifie les positions du joueur par rapport à de nouvelles positions. Les 4 points du joueur formant sa hitbox sont vérifiés par rapport une valeur hexadécimale (voir le chapitre 7). Chaque carreau ayant la propriété `passable` est passé au crible sur la fonction privée `pointIsPassableInTile` pour la praticabilité. Il suffit qu'un point ne passe pas pour renvoyer le booléen `false` et bloquer le joueur.

## Mouvement, gravité et saut

C'est le modèle `Game_Character` qui se charge du mouvement, de la gravité et du saut selon le chapitre 5.

**Figure 12-4**

Saut et gravité dans le jeu



### Mouvements

```

Class.create("Game_Character", {
    map: null,
    currentState: "",
    a: 0,
    speed: 4,
    x: 0,
    y: 0,
    dir: "right",
    _gravity: {
        power: 10,
        velocity: 0
    },
    _jump: {

```

```
        power: 10,
        velocity: 1
    },
    initialize: function(id, width, height, x, y, map) {
        this.x = x;
        this.y = y;
        this.id = id;
        this.width = width;
        this.height = height;
        this.map = map;
    },
    move: function(dir) {
        this.dir = dir;
        this.a += .05;
        if (this.a >= 1) {
            this.a = 1;
        }
        var speed = this.speed * this.a,
            x = this.x;
        switch (dir) {
            case "left":
                x -= speed;
                break;
            case "right":
                x += speed;
                break;
        }
        if (this.map.isPassable(this, x, this.y)) {
            this.x = x;
        }
        return this.x;
    },
    jump: function(state) {
        if (state && this.currentState == "platform") {
            this.currentState = "jumping";
        }
        else if (!state && this.currentState == "jumping") {
            this._jump.velocity = 1;
            this.currentState = "godown";
        }
    },
    jumpUpdate: function() {
        var velocity = this._jump.velocity, new_y = this.y;
        if (this.currentState == "jumping") {
            velocity -= .05;
            if (velocity <= 0) {
                velocity = 1;
                this.currentState = "godown";
            }
        }
        else {
```

```

        new_y -= this._jump.power * velocity;
    }
    if (this.map.isPassable(this, this.x, new_y)) {
        this._jump.velocity = velocity;
        this.y = new_y;
    }
}
return this.y;
},
gravityUpdate: function() {
    var velocity = this._gravity.velocity, new_y = this.y;
    if (this.currentState != "jumping") {
        velocity += .05;
        if (velocity >= 1) {
            velocity = 1;
        }
        new_y += (this._gravity.power * velocity);
        if (this.map.isPassable(this, this.x, new_y)) {
            this.y = new_y;
            this._gravity.velocity = velocity;
        }
        else {
            this.currentState = "platform";
            this._gravity.velocity = 0;
        }
    }
    return this.y;
}
});

```

La méthode `move` change la position `x` du personnage avec une accélération au commencement du mouvement. Bien sûr, nous testons si la prochaine position est possible en vérifiant la praticabilité calculée dans le modèle `Game_Map`.

Le saut a une décélération, diminue la propriété `y` si le carreau autorise le passage du personnage et change son statut pour indiquer qu'il retombe et exécuter la méthode `gravityUpdate`. Cette dernière procède de la manière inverse : accélération, augmentation de la valeur de `y` jusqu'à ce qu'une plate-forme soit touchée. L'état change pour indiquer que le personnage est sur une plate-forme, éviter d'appliquer la gravité et donner la possibilité de ré-effectuer un saut.

## Effectuer un mouvement selon une touche

Nous devons appeler le modèle pour effectuer le saut à l'appui d'une touche.

### Gestion de la touche Haut

```
canvas.Input.press(Input.Up, function() {
    self.game_player.jump(true);
```

```

    });

    canvas.Input.keyUp(Input.Up, function() {
        self.game_player.jump(false);
    });

    canvas.Input.keyUp(Input.Right, function() {
        anim_right.stop();
        self.game_player.moveClear();
        self.player.drawImage("gripe_stand_right");
    });

    canvas.Input.keyUp(Input.Left, function() {
        anim_left.stop();
        self.game_player.moveClear();
        self.player.drawImage("gripe_stand_left");
    });
}

```

Quand la touche *Haut* est enfoncee, la méthode `jump` du modèle `Game_Character` est exécutée changeant l'état du personnage pour effectuer un saut. En revanche, si cette touche est relâchée, nous terminons le saut même si celui ci était en cours.

Enfin, au relâchement des touches *Droite* et *Gauche*, nous arrêtons l'animation des déplacements et affichons une image fixe à la place.

La méthode `moveClear` propre à `Game_Player` réinitialise la valeur de l'accélération.

#### Réinitialisation de la valeur d'accélération

```

Class.create("Game_Player", {
    moveClear: function() {
        this.a = 0;
    }
}).extend("Game_Character");

```

La méthode `render` de la scène a cette apparence au final :

#### Le rendu du jeu

```

render: function(stage) {
    if (!this.game_player) {
        return;
    }

    var input = {
        "left": Input.Left,
        "right": Input.Right
    };
    for (var key in input) {
        if (canvas.Input.isPressed(input[key])) {
            this.player["x"] = this.game_player.move(key);
        }
    }
}

```

```
        }

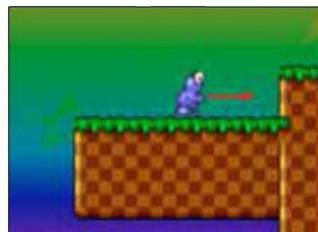
        if (canvas.Input.isPressed(Input.Up)) {
            this.game_player.jumpUpdate();
        }

        this.scrolling.update();
        this.player.y = this.game_player.gravityUpdate();
        stage.refresh();
    }
```

Si la touche *Gauche* ou *Droite* est enfoncée, nous recherchons la valeur de la nouvelle position et l'appliquons à la propriété *x* du Sprite.

**Figure 12-5**

Mouvement sur la droite à l'appui de la touche Droite



Pour la touche du saut et la gravité, nous mettons à jour la position *y* du Sprite en appelant les méthodes créées dans le modèle *Game\_Character*.



# 13

## Configurer le serveur pour le multijoueur

---

Avant de créer un jeu multijoueur, nous devons configurer le serveur pour faire tourner Node.js.

Node.js sert à créer des applications en temps réel. Il est basé sur le moteur Javascript V8. Vous utilisez donc le Javascript aussi bien côté client que côté serveur.

Socket.io est un module de Node.js permettant de partager des données en temps réel quelle que soit la plate-forme (navigateurs, smartphones, tablettes tactiles, etc.).

Socket.io utilise différents modes de transport :

- WebSocket;
- Adobe® Flash® Socket;
- AJAX long polling;
- AJAX multipart streaming;
- Forever Iframe;
- JSONP Polling.

#### REMARQUE Linux

Vous devez avoir un serveur sous Linux pour les commandes qui suivent.

## Utiliser le serveur via SSH

Vous possédez votre propre serveur ou un serveur dédié. Il vous faut désormais vous connecter et installer Node.js. Pour cela, connectez-vous avec le programme nommé Putty utilisant le protocole SSH.

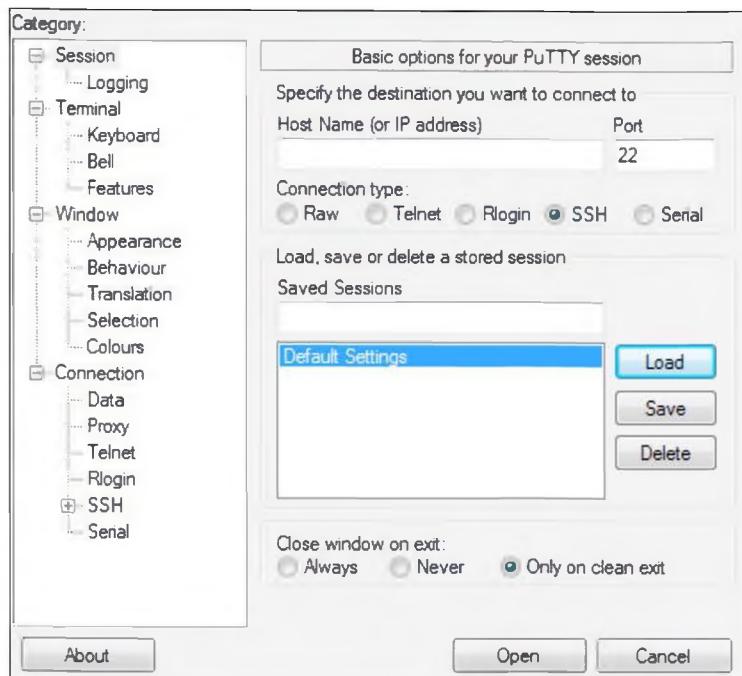
- 1 Entrez l'adresse IP de votre serveur dans *IP Adress*.
- 2 Laissez le port 22 et la connexion SSH.
- 3 Pour les prochaines fois, sauvegardez la session en rentrant un nom personnalisé et cliquez sur *Save*.
- 4 Ouvrez le Shell en cliquant sur *Open*.
- 5 Entrez le nom d'utilisateur. Dans le cas où vous êtes un super administrateur, mettez *root*.
- 6 Entrez le mot de passe. Par sécurité, rien ne s'affiche à l'écran.

#### RAPPEL Commandes Linux

Voici quelques commandes utiles pour Linux :

- `cd` : changer de répertoire;
- `ls` : afficher les fichiers du répertoire courant;
- `pwd` : chemin courant;
- `cp` : copier un fichier;
- `rm` : supprimer un fichier ou dossier (ajoutez `-rf` pour une suppression récursive des sous-dossiers);
- `chmod` : changer les permissions d'un dossier.

**Figure 13-1**  
Se connecter avec Putty



## Télécharger Node.js

Vous pouvez télécharger le paquet sur <http://nodejs.org/download/> ou bien saisir la ligne suivante dans le Shell :

### Télécharger Node.js

```
git clone git://github.com/joyent/node.git
```

Tapez la commande suivante si git n'est pas installé :

```
apt-get install git-core
```

Allez dans le dossier contenant les fichiers de Node.js (`cd node`) puis commencez l'installation :

Procéder à l'installation

```
./configure && make && make install
```

**Figure 13-2**

Erreur lors de l'installation de Node.js



#### ERREUR Installation Node.js

Si vous obtenez cette erreur, tapez les lignes suivantes puis recommencez l'installation de Node.js :

```
| aptitude install build-essential  
| apt-get install libssl-dev
```

## Installer NPM

NPM signifie *Node Packaged Modules*. Il s'agit d'une bibliothèque de modules pour Node.js qui étendent les fonctionnalités dans notre application côté serveur.

#### Installation de NPM

```
| curl -k https://npmjs.org/install.sh " sh
```

#### EN PRATIQUE Si CURL n'est pas installé

Tapez la ligne suivante :

```
| apt-get install curl
```

## Installer Socket.io

#### Installation de Socket.io

```
| npm install socket.io
```

Ceci va créer un dossier `node_modules`. Lorsque vous démarrez votre serveur avec Node.js, vérifiez que ce dossier soit le parent du serveur ou le même chemin. Par exemple, si le fichier pour le serveur se trouve sur `/apps/node/server.js`, le dossier des modules doit être dans l'un des emplacements suivants :

- `/apps/node/node_modules`
- `/apps/node_modules`
- `/node_modules`

## Tester l'installation

Vous pouvez tester le bon fonctionnement de votre installation avec le code présent sur le site de Socket.io.

► <http://socket.io>

### ERREUR Ça ne fonctionne pas!

Essayez le port « 8333 » au lieu de « 80 » si ce dernier est déjà utilisé (un message vous en avertit).

Démarrez votre serveur : `node server.js`.

Ici, nous avons pris le nom `server.js`. Bien entendu, vous devez saisir le chemin vers le fichier contenant le code du serveur.

### ASTUCE Charger des fichiers facilement

Au lieu d'utiliser des commandes pour charger ou télécharger des fichiers sur le serveur, utilisez un FTP comme Filezilla et ajoutez un site dans les *Gestionnaire des sites*. Mettez le protocole en *SFTP* pour utiliser le transfert des fichiers en SSH. Vous pouvez désormais manipuler le système de fichiers aisément.



# 14

## Utilisez Node.js pour votre jeu multijoueur en temps réel

---

Un jeu multijoueur en temps réel se base sur Node.js et Socket.io. Comment réaliser le code côté serveur et interagir avec le client ?

## Comment fonctionne Socket.io ?

Socket.io est une extension de Node.js utilisant les WebSockets ou d'autres transports selon la plate-forme utilisée.

Socket.io fonctionne sur la base d'événements interactifs comme le clic, le déplacement de la souris, etc. Dès qu'un envoi est effectué, l'événement du serveur est déclenché.

Avant tout, il faut connecter le joueur au serveur :

### Test de Socket.io côté serveur

```
var app = require('http').createServer(), ①
    io = require('socket.io').listen(app); ②

app.listen(8333); ③

io.sockets.on('connection', function (socket) { ④
    socket.emit ⑤('news', { hello: 'world' });
    socket.on ⑥('my other event', function (data) {
        console.log(data);
    });
});
```

Ici, nous effectuons plusieurs étapes :

- ① Nous insérons le module `http` présent dans Node.js pour créer un serveur HTTP.
- ② Nous insérons le module `socket.io`.
- ③ Le serveur écoute les paquets sur le port 8333.
- ④ Un événement `connection` est déclenché quand un client se connecte. Le paramètre `socket` est propre à l'utilisateur.
- ⑤ Nous envoyons des données à l'utilisateur avec `emit`.
- ⑥ La méthode `on` permet de définir les événements à déclencher.

Le code Javascript côté client est le suivant :

### Test de Socket.io côté client

```
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io.connect('http://127.0.0.1:8333'); ⑦
    socket.on('news' ⑧, function (data) {
        console.log(data);
        socket.emit('my other event', { my: 'data' }); ⑨
    });
</script>
```

- ⑦ Nous réalisons une connexion au serveur en indiquant l'adresse IP et le port.
- ⑧ Nous réceptionnons les données de l'événement `news`.

- ➉ Nous émettons des données vers le serveur.

## Fonctionnement dans CanvasEngine

Vous avez le principe des événements avec Socket.io. Dans CanvasEngine, il faut distinguer plusieurs parties :

- scène ;
- vue ;
- modèle.

Dans les premiers chapitres, nous avons vu comment réaliser un jeu en séparant les différentes parties sans pour autant en avoir réellement l'utilité. En fait, cela vous préparait à concevoir un jeu multijoueur.

Les événements existent toujours sauf qu'ils sont camouflés par des classes, afin de privilégier la programmation orientée objet.

Avant tout, installez le modèle de CanvasEngine qui fait une surcouche de Socket.io de façon à utiliser des classes plutôt qu'un système événementiel :

### Installation du module CanvasEngine

```
npm install canvasengine
```

Vous avez désormais un fichier nommé `server.js` possédant le code suivant :

#### Code initial côté serveur

```
var CE = require("canvasengine" ①).listen(8333); ②
CE.Model.init("Main" ③, {
    initialize: function() {
    }
});
```

➊ Nous importons le module `canvasengine`. Il contient la déclaration des classes de type « modèle » sur le serveur et plusieurs modules dont Socket.io.

➋ Le jeu écoute les paquets sur le port 8333.

➌ Nous définissons la classe principale appelée lors de la connexion du joueur : son nom est `Main` et la méthode `initialize` est son constructeur.

## Définir les événements

Comment définir les événements dans notre classe ? Nous ajoutons un paramètre déclarant les noms des événements correspondant aux méthodes de la classe :

**Définition d'un événement côté serveur**

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", ["start"], {

    initialize: function() {

    },
    start: function() {
        this.scene.emit("load", "Go !");
    }
});
```

Voyons le code Javascript Complet côté client :

**Définition d'un événement côté client**

```
var Model = io.connect('http://127.0.0.1:8333');
var canvas = CE.defines("canvas").
    ready(function(ctx) {
        canvas.Scene.call("MaScene");
    });

canvas.Scene.new({
    name: "MaScene",
    model: Model,
    events: ["load"], ②
    ready: function(stage) {
        this.model.emit("start"); ①
    },
    load: function(text) {
        console.log(text);
    }
});
```

Contrairement à un simple jeu, le modèle ne sert qu'à envoyer des données au serveur. Nous appelons la méthode `start` ①, présente dans la classe `Main` définie dans le serveur. De la même manière, des données sont envoyées à la scène et la méthode `load` est appelée.

## Composer la structure du jeu

CanvasEngine impose une structure permettant d'avoir un modèle côté serveur et une scène côté client. Cela ne signifie pas pour autant que votre jeu aura une structure bien fondée.

Il faut retenir que la scène ne fait que récupérer des données du modèle et les afficher sur le `canvas`. Ainsi, ne créez pas des classes possédant des propriétés inutiles pour la scène.

Prenons un exemple. Vous souhaitez afficher un personnage sur une carte. Ce personnage est caractérisé par des positions X et Y, une direction, une apparence et un état (debout, marche, etc.) pour les animations.

- 1 Le modèle calcule les positions du personnage. S'il se déplace, il appliquera l'état « marche » et sa direction.
- 2 Le modèle envoie à la scène (i.e. le serveur envoie au client) les valeurs des positions, de la direction et de l'état.
- 3 La scène dispose d'une classe des Sprites pour afficher des images ou animations sur l'écran. Lors de la récupération des données, elle met à jour les éléments sur l'écran. Il est inutile de stocker les paramètres puisque le serveur possède déjà l'information ; ils ne serviront qu'à mettre à jour les images.

Nous avons vu précédemment comment définir les événements d'une classe nommée `Main`. Cependant, est-il logique d'insérer des méthodes pour le déplacement d'un joueur, le chargement d'une carte ou l'envoi d'un message, alors qu'il n'y a pas de lien entre ces actions ? Non, ne mélangeons pas tout ! Dans un jeu, il faut définir plusieurs classes pour gérer les différentes parties de celui-ci. Ainsi, nous pouvons avoir :

- une classe pour la gestion des joueurs ;
- une classe pour la gestion des objets sur le niveau ou la carte ;
- une classe pour la gestion des cartes ;
- etc.

Nous allons créer un module dans Node.js pour définir les classes de notre jeu.

## Créer des modules dans Node.js

Node.js charge des modules avec la fonction `require`. Le module est un fichier avec extension `.js` (fichier Javascript interprété), `.json` ou `.node` (extension de Node.js chargé avec `dlopen`).

Le chemin du fichier peut être absolu (commençant par `/`) ou relatif (commençant par `./`). Si aucun des deux symboles n'est indiqué, le module sera cherché dans le dossier `node_modules`.

Node.js comporte plusieurs modules compilés – décrits dans la documentation du site. Ils sont définis dans le répertoire `lib/` des sources du programme. Ces modules sont chargés grâce à leur identifiant. Par exemple, `require('http')` retourne le module HTTP même s'il en existe déjà un de même nom.

### Créer un modèle

Nous allons créer une classe pour la gestion des cartes du jeu. Elle servira à charger une carte, obtenir des informations comme largeur ou hauteur, connaître les propriétés des carreaux (identifiants, praticabilité), etc.

Créons à la racine un fichier nommé `Game_Map.js` et ajoutons-y les dépendances.

#### Exporter un module

```
exports.Class = function(CE) {
    // La classe sera insérée ici
};
```

Point élémentaire à retenir, il faut utiliser l'objet `exports` pour exporter une valeur du module. La variable fait partie du système de modules de Node.js. Ainsi, vous pourrez récupérer la fonction précédente de cette façon :

#### Récupération du module

```
var GameMap = require("./Game_Map").Class
```

Pour le moment, restons dans le modèle. Le paramètre `CE` sert pour définir un nouveau modèle.

#### Création du modèle

```
exports.Class = function(CE) {
    CE.Model.create("Game_Map" ①, ["load"] ②, {
        initialize: function() { ③
            },
            load: function() { ④
                }
            }).attr_reader([
                "map_id" ⑤
            ]);
        return CE.Model.new("Game_Map");
};
```

- ① Le modèle est nommé `Game_Map`.
- ② Nous définissons le tableau des noms des événements.
- ③ Nous créons les méthodes (ainsi que le constructeur `initialize`).
- ④ Nous déclarons les attributs. Ici `map_id` est l'identifiant de la carte.

**REMARQUE** Les attributs

Les attributs sont les propriétés de la classe. CanvasEngine intègre leur déclaration par l'intermédiaire des méthodes `attr_accessor`, `attr_reader` et `attr_writer` (noms inspirés du Ruby). Ces dernières respectent le principe de l'encapsulation en définissant trois niveaux de visibilité :

- `attr_accessor` : attribut accessible et modifiable;
  - `attr_reader` : attribut seulement accessible;
  - `attr_writer` : attribut seulement modifiable.
- Ces attributs sont à rapprocher de `get` et `set`. Par exemple, une propriété nommée `foo` possède un accesseur :

```
var val = this.foo.get();
```

- et un modificateur :

```
this.foo.set(val);
```

- Par ailleurs, dans la classe elle-même, on peut accéder à une propriété privée avec le préfixe `_` :

```
this._foo = val;
```

Ici, quand la scène appelle l'événement `load`, la méthode du même nom de la classe `Game_Map` est exécutée. L'intérêt est de travailler dans cette classe et avec ses propriétés.

Le modèle étant créé, nous allons le déclarer dans la classe principale.

**Déclaration du modèle dans la classe principale**

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", {
    initialize: function() {
        this.GameMap = require("./Game_Map").Class(CE);
    }
});
```

La méthode `listen` (en réalité, une surcouche de la méthode `listen` de `http.createServer`) écoute le port 8333. La méthode `init` appartient à la classe `CE.Model` et ajoute un écouteur à l'événement `connection` de Socket.io.

La variable `CE` sert à importer le module du même nom, que nous injectons dans les paramètres de notre classe `Game_Map`. Nous effectuons deux actions en une ligne :

1 Importation du module.

2 Exécution de la fonction qui crée la classe pour l'affecter à l'instance `GameMap`.

La variable `GameMap` est une instance de la classe `Main`, dont elle peut utiliser les méthodes. En revanche, si vous créez un second module (`Game_Player` par exemple) et souhaitez y utiliser les propriétés de `Game_Map`, cela sera impossible.

L'astuce est de rendre ces variables communes à tous les modules.

Node.js propose une variable nommée `global`. C'est un objet disponible pour tous les modules. L'instance `GameMap` sera transformée de la façon suivante :

**Variable globale**

```
| global.GameMap = require("./Game_Map").Class(CE);
```

## Base de données

Nous allons stocker dans un fichier à part toutes les données fixes du programme. Nous utiliserons le format JSON comme pour les niveaux ou les cartes.

### Schéma

Formez une collection des objets de votre jeu :

**Schéma des objets**

```
{  
  "1": {  
    "name": "Potion",  
    "hp": 5  
  },  
  "2": {  
    "name": "Mega Potion",  
    "hp": 15  
  }  
}
```

Les objets possèdent trois propriétés :

- `id` : l'identifiant pour récupérer les données dans le modèle ;
- `name` : le nom pour l'affichage ;
- `hp` : le nombre de points de vie que le joueur récupérera.

De la même manière, voici comment les adversaires du jeu peuvent être schématisés :

**Schéma des adversaires**

```
{  
  "1": {  
    "atk": 10,  
    "def": 2,  
    "hp": 100  
  },  
  "2": {  
    "atk": 8,  
    "def": 5,  
    "hp": 80  
  }  
}
```

Un adversaire possède quatre propriétés :

- `id` : identifiant comme pour les objets ;
- `atk` : une valeur d'attaque ;
- `def` : une valeur de défense ;
- `hp` : le nombre maximal de points de vie.

Les propriétés sont utilisées pour les calculs.

Dans les deux exemples, les données sont séparées. Nous allons les fusionner dans un seul fichier JSON :

#### Ensemble des schémas

```
{
  "items": {
    //Données des objets
  },
  "ennemis": {
    //Données des adversaires
  }
}
```

### Données dans le modèle

Les données sont stockées dans `Data/Database.json`. Pour les obtenir, seulement en lecture, nous utilisons la méthode habituelle `getJSON`.

#### Affectation de la base de données à une variable globale

```
global.CE.getJSON("Data/Database.json", function(data) {
  global.data = data;
});
```

#### DÉTAIL getJSON côté serveur

La méthode `getJSON` réalise une requête Ajax. Cependant, côté serveur, elle utilisera le système de fichiers de Node.js.

#### Lecture des données

```
for (var id in global.data.items) {
  console.log(global.data.items[id]);
}
```

Bien sûr, il est recommandé de regrouper dans une classe les méthodes de gestion des données. Créez un fichier `Game_Items` pour un nouveau module.

```
Création du modèle pour la gestion des objets du jeu
exports.Class = function(CE) {
    CE.Model.create("Game_Items", {
        initialize: function() {
            this.items = global.data.items;
        },
        get: function(id) {
            return this.items[id];
        }
    });
    return CE.Model.new("Game_Items");
};
```

## Gérer les connexions et déconnexions

Pour se connecter au serveur, le modèle affecté à la scène est tout simplement Socket.io :

### Connexion au serveur

```
var model = io.connect("http://IP:8333");
canvas.Scene.new({
    name: "Scene",
    model: model,
    ready: function(stage) {
    }
});
```

Lorsque nous faisons référence au modèle dans la scène, nous appelons en réalité les méthodes de Socket.io. Pour intercepter la connexion établie, ajoutez un événement nommé `connect` :

### Événement dans la scène

```
canvas.Scene.new({
    name: "Scene",
    events: ["connect"],
    model: model,
    ready: function(stage) {

    },
    connect: function() {

    }
});
```

**DÉTAIL Autre méthode**

Pour savoir si le joueur est connecté, vous pouvez aussi utiliser la méthode traditionnelle de Socket.io :

```
this.model.on("connect", function() {
  // Code
});
```

Côté serveur, la connexion est captée dans l'initialisation de la classe principale. Cette dernière possède une méthode `disconnect` qui s'exécute dès que le joueur se déconnecte.

**Déconnexion côté serveur**

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", {
  initialize: function() {
    // Connexion du joueur
  },
  disconnect: function() {
    // Déconnexion du joueur. Appel de la scène
    this.scene.disconnect();
  }
});
```

Sur la scène, côté client, la déconnexion se déroule de la même manière que la connexion. Nous ajoutons un événement nommé `disconnect`.

**Déconnexion côté client**

```
canvas.Scene.new({
  name: "Scene",
  events: ["connect", "disconnect"],
  model: model,
  ready: function(stage) {

  },
  connect: function() {

  },
  disconnect: function() {

  }
});
```

**Données communes**

Lorsque le joueur se connecte au jeu, des données peuvent déjà être présentes. Les données sont donc communes et gardées en mémoire sur le serveur. La procédure est

extrêmement simple : il suffit de créer des variables globales et de les lire dans la classe contenant les données communes.

```
var users = [];

var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", {

    initialize: function() {

    },
    addUser: function(data) {
        users.push(data); // data : {name: "Nom", pseudo: "pseudo"}
    },
    getUsers: function() {
        this.scene.emit("users", users);
    }
});
```

La variable globale `users` est un tableau contenant tous les utilisateurs connectés. Les éléments sont des objets (`{name: «Nom», pseudo: «pseudo»}` par exemple).

La méthode `addUser` ajoute donc un utilisateur et la méthode `getUsers` récupère tous les utilisateurs pour afficher leur nom sur la scène. La variable `users` garde les données tant que le serveur est en marche et est commune à tous les utilisateurs.

## Partage des données entre joueurs

Le but ultime d'un jeu multijoueur est de partager les données entre joueurs. Dans la scène, nous appelons une méthode sur le modèle présent sur le serveur.

Admettons que nous envoyons une donnée après un clic sur un élément :

### Envoi d'une donnée après un clic sur l'élément

```
var self = this,
    el = this.createElement(50, 50);
el.on("click", function() {
    self.model.emit("mymethod", {type: "foo"});
});
```

Le code, contenu dans la méthode `ready` de la scène, envoie la donnée `foo` à la méthode `mymethod` sur le serveur :

### Réception de la donnée côté serveur

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", ["mymethod"], {

    initialize: function() {
```

```
    },
    mymethod: function(data) {
        this.scene.emit("myresponse", data);
    }
});
```

`mymethod` est donc exécutée et renvoie la donnée aux utilisateurs du jeu en appelant la méthode `myresponse` de la scène :

#### Réception de la donnée côté client

```
canvas.Scene.new({
    name: "Game",
    model: Model,
    events: ["myresponse"],
    ready: function(stage) {
        // Code
    },
    myresponse: function(data) {
        console.log(data.type);
    }
});
```

La méthode `myresponse` affichera la donnée, soit « foo ». Mais comment cibler seulement un groupe de personnages ?

Dans plusieurs jeux, les données ne sont pas envoyées à tous les utilisateurs connectés. Il est par exemple possible que seuls les joueurs d'une carte ou d'un niveau soient concernés. Dans ce cas, nous devons joindre le joueur à une « room ». Ainsi, en envoyant des données ultérieurement, seules les personnes dans la « room » les recevront :

#### Joindre et quitter une room

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", ["openMap", "exitMap"], {

    initialize: function() {

    },

    openMap: function(data) {
        this.scene.join("nom_de_la_room");
    },

    exitMap: function(data) {
        this.scene.leave("nom_de_la_room");
    }
});
```

Le nom à attribuer peut très bien être le nom du niveau ou d'un joueur. Tout dépend du concept de votre jeu. Pour quitter la « room », nous utilisons la méthode `leave`. Voyons comment envoyer une donnée :

#### Envoyer une donnée aux joueurs de la room

```
var CE = require("canvasengine").listen(8333);
CE.Model.init("Main", ["hello", "openMap", "exitMap"], {
    initialize: function() {
    },
    hello: function() {
        this.scene.broadcast.to("nom_de_la_room", "mymethod", {msg: "Hello"});
    },
    openMap: function(data) {
        this.scene.join("nom_de_la_room");
    },
    exitMap: function(data) {
        this.scene.leave("nom_de_la_room");
    }
});
```

Nous utilisons la méthode `broadcast` pour envoyer le message « Hello » aux personnes dans la « room ».

La réception dans la scène ne change pas.

## Sauvegarde et chargement avec Mongoose

### Installation de MongoDB et Mongoose

Mongoose est un module de Node.js basé sur le gestionnaire de bases de données MongoDB. Ce dernier est un système de stockage orienté documents n'utilisant pas le SQL. Dans un premier temps, installez MongoDB :

- 1 Allez sur <http://www.mongodb.org/downloads> pour télécharger les paquets ou tapez la ligne suivante sur Linux : `sudo apt-get install mongodb-10gen`
- 2 Démarrez MongoDB : `sudo /etc/init.d/mongodb start`

### WINDOWS Démarrer MongoDB

Sur Windows, ouvrez l'invite de commande et accédez au dossier `bin` du fichier téléchargé avec la commande `cd`.

Tapez `mongo` pour démarrer le service. En cas de problème, tapez `mongod` pour avoir un retour de l'échec. Il se peut que MongoDB vous demande de créer les dossiers `data/db` à la racine du disque.

Ensuite, installez le module Mongoose :

#### Installation de Mongoose

```
npm install mongoose
```

## Connexion à la base et schéma

Nous importons le module dans notre projet (côté serveur) et établissons une connexion :

#### Connexion à la base de données

```
var mongoose = require('mongoose'),  
    db = mongoose.createConnection('localhost', 'test');
```

Nous créons un objet `db` qui effectue la connexion sur la base de données `test` et tourne sur le serveur `localhost`. Pour définir les entrées, nous initialisons un schéma :

#### Création du schéma

```
var schema = mongoose.Schema({ name: 'string' });  
var User = db.model('User', schema);
```

Notre schéma portera seulement le nom de l'utilisateur compilé dans le modèle `User` pour effectuer des requêtes.

## Sauvegarder des données

Les données sont ensuite sauvegardées en utilisant la méthode `save`. Si des erreurs surviennent, traitez-les dans la condition.

#### Insertion d'une donnée

```
var data = new User({ name: 'Sam' });  
data.save(function (err) {  
    if (err) {  
        }  
});
```

Pour mettre à jour les données, utilisez la méthode `update`.

### Mise à jour d'une donnée

```
User.update({ name : 'Sam' } 1, {name: "Jim"} 2, function (err) 3 {
  if (err) { throw err; }
});
```

Les paramètres sont les suivants :

- **1** condition ;
- **2** la mise à jour ;
- **3** fonction de rappel.

## Chargement

Pour charger, il faut reprendre le modèle créé (composé du schéma). Ensuite, appliquez la méthode `find` :

### Recherche d'une donnée

```
User.find({name: /Sam/i}, null, function (err, data) {
  console.log(data); // object
});
```

Le premier paramètre cherche le nom « Sam » dans le champ `name`. La fonction de rappel récupère les erreurs mais surtout les données renvoyées sous la forme d'un objet.

Le deuxième paramètre indique les champs à rechercher. C'est une chaîne de caractères où les champs sont séparés d'un espace.

Par exemple, imaginons que nous avons le champ `friends` dans le schéma du modèle :

### Recherche d'une donnée dans plusieurs champs

```
User.find({name: /Sam/i}, 'name friends', function (err, data) {
  console.log(data); // object
});
```

Ici, le nom sera recherché dans les champs `name` et `friends`.

### EXPRESSION RÉGULIÈRE Les drapeaux

Une expression régulière dans Javascript est comprise entre deux barres obliques et peut se terminer par un drapeau après la dernière barre oblique :

- `g` : Rechercher global (sur plusieurs lignes)
- `i` : Ignorer la casse typographique
- `m` : Traiter les caractères du début à la fin (^ et \$) sur plusieurs lignes délimitées par \n ou \r

# 15

## Intégration du jeu à un réseau social : Facebook

---

Intégrer un jeu à un réseau social nécessite une authentification et une autorisation de l'application. Ce chapitre explique cette procédure sur le réseau social le plus connu : Facebook.

## Déclaration du jeu dans Facebook

### IDENTITÉ N'est pas développeur Facebook qui veut...

Si c'est la première fois que vous déclarez une application sur Facebook, il vous faut d'abord confirmer votre identité avec un téléphone portable (en rentrant le code que Facebook vous envoie par SMS) ou par l'intermédiaire d'un numéro de carte de crédit.

Avant toute procédure, il faut déclarer la nouvelle application dans Facebook pour utiliser les fonctionnalités :

- 1 Allez sur lien <http://developers.facebook.com/apps>.
- 2 Cliquez sur le bouton *Créer une application*. Une fenêtre apparaît :

**Figure 15-1**

Créer une nouvelle application dans Facebook



**Tableau 15-1. Explications des champs**

Champ	Explication
Nom de l'application	Nom qui apparaîtra dans la recherche, les favoris et autres
App Namespace	Identifiant qui s'affichera dans l'URL de la manière suivante : apps.facebook.com/mysocialgaming
Web Hosting	Si vous souhaitez faire héberger votre application. Laissez décoché.

- 3 Validez et saisissez le texte du contrôle de sécurité.
- 4 La suite vous invite à donner plus de détails à propos de votre application :

**Figure 15-2**

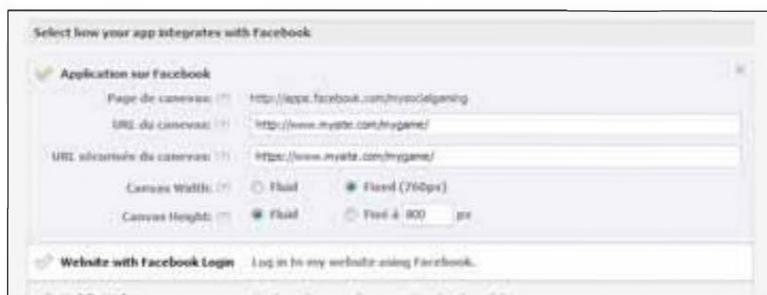
Informations supplémentaires sur l'application

Tableau 15-2. Explication des champs

Champ	Explication
App Domains	Autoriser l'authentification sur un domaine ou sous domaine
Catégorie	Catégorie de votre application. Ici, nous sélectionnons Jeux
Hosting URL	Espace web attribué (ignoré)
Mode Bac à Sable	Activez-le pour vous assurer que les développeurs y ont accès lors de la programmation.

5 Sélectionnez *Application sur Facebook* pour rentrer les liens vers le jeu :

**Figure 15-3**  
Liens vers le jeu



6 Mettez le lien dans *Mobile Web*.

#### À RETENIR

- Les liens doivent se terminer par / si vous faites référence à une page index.
- Le nom de domaine utilisé, *App Domains*, doit être présent dans les liens.

7 Cliquez sur *Enregistrer les modifications*.

## Authentification et autorisation

Pour obtenir une autorisation, nous devons utiliser le SDK en PHP de Facebook.

#### OUTIL Télécharger le SDK en PHP de Facebook

Téléchargez le fichier ZIP sur

► <https://github.com/facebook/facebook-php-sdk>

**Figure 15-4**

Téléchargement du SDK de Facebook sur Github



Une page demandera l'autorisation de l'application dans Facebook. Créez donc une page *index.php* dans un dossier externe au jeu.

#### Demande d'autorisation à Facebook

```
<?php
require 'src/facebook.php'; ①

define('APP_ID', '386722101395435'); ②
define('APP_SECRET', '5e2c6440583743a47b4e*****'); ③
define('APP_NAMESPACE', 'mysocialgaming'); ④
define('APP_URL', 'http://apps.facebook.com/' . APP_NAMESPACE . '/'); ⑤

$scope = 'email,publish_actions'; ⑥

$facebook = new Facebook(array( ⑦
    'appId'  => APP_ID,
    'secret' => APP_SECRET,
));

$user = $facebook->getUser(); ⑧

if (!$user) { ⑨
    $loginUrl = $facebook->getLoginUrl(array( ⑩
        'scope' => $scope,
        'redirect_uri' => APP_URL,
    ));
    print('<script> top.location.href=\'' . $loginUrl . '\'</script>'); ⑪
}
?>
```

Le dossier *src* de Facebook SDK se trouve au même endroit que cette page.

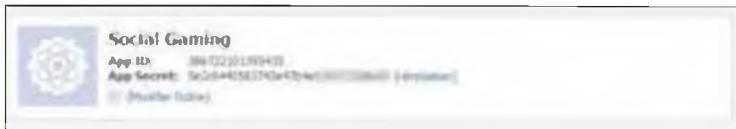
- 1 Nous l'importons avec `require` ①.
- 2 Nous créons quatre constantes :
  - `APP_ID` est l'identifiant de l'application ②.

- APP\_SECRET est une valeur hexadécimale privée **3**.
- APP\_NAMESPACE est le nom de domaine dans l'URL **4**.
- APP\_URL est le lien complet vers le jeu Facebook **5**.

APP\_ID et APP\_SECRET sont affichées en en-tête de la page de réglage de l'application.

**Figure 15-5**

Pour définir APP\_ID et APP\_SECRET



- 3 La variable \$scope indique les informations à récupérer (e-mail) et les actions (publier au nom de l'utilisateur) **6**.
- 4 Nous créons une nouvelle instance de la classe Facebook avec les identifiants en constante **7**.
- 5 Nous récupérons l'identité de l'utilisateur. Si ce dernier a déjà autorisé l'application, son identifiant est renvoyé **8**.
- 6 Le cas échéant, c'est la valeur 0 qui est renvoyée et nous rentrons dans la condition **9**.
- 7 Nous créons l'URL pour l'autorisation de l'application en désignant les permissions (variable \$scope) et le lien de l'application **10**.
- 8 Nous redirigeons vers l'URL créée. Puisque le jeu est chargé dans une iframe, Facebook ne peut pas faire de redirection côté serveur, alors que nous avons besoin de rediriger le cadre parent. Pour cette raison, nous faisons une redirection côté client en Javascript avec top.location **11**.

**Figure 15-6**

Demander l'autorisation pour jouer sur le jeu



Après l'autorisation, si l'internaute revient sur la page créée, il sera automatiquement redirigé vers Facebook pour jouer.



# 16

## Implémentation de la partie Social Gaming

---

Exploitez dans votre jeu les données du réseau du joueur et poussez votre Gameplay sur le Social Gaming.

Le jeu se trouve sur le lien déclaré lors de la déclaration dans Facebook (voir chapitre 15). En fait, il vous suffit de créer un jeu en HTML5 et Javascript sur votre propre site. Ensuite, toutes les manipulations ou récupérations du compte de l'utilisateur sont prises en charge par le SDK du réseau social.

Ce SDK nous donne les moyens d'inviter des amis à jouer, de partager des informations, d'intégrer des objectifs ou scores et d'avoir une monnaie virtuelle : tout ce dont nous avons besoin dans un jeu de Social Gaming !

## Intégration du jeu HTML5

L'utilisateur a autorisé l'application à récupérer des données le concernant. Avant de les utiliser dans le jeu, nous devons :

- 1 intégrer le SDK de Facebook;
- 2 initialiser;
- 3 savoir si l'utilisateur est bien connecté à son compte.

### Intégration du SDK et initialisation

Facebook injecte du Javascript dans la page. Il permet d'utiliser le SDK et se charge de façon asynchrone pour ne pas bloquer la page.

#### Intégration du SDK de Facebook

```
(function(d){
  var js, id = 'facebook-jssdk', ref = d.getElementsByTagName('script')[0];
  if (d.getElementById(id)) {return;}
  js = d.createElement('script'); js.id = id; js.async = true;
  js.src = "//connect.facebook.net/fr_FR/all.js";
  ref.parentNode.insertBefore(js, ref);
}(document));
```

Ce code est disponible dans la documentation de Facebook. Copiez-coller-le dans votre page.

#### Initialisation

```
window.fbAsyncInit = function() {
  FB.init({
    appId      : 'ID_DE_L_APPLICATION',
    status     : true,
    cookie    : true
  });
};
```

La méthode `fbAsyncInit` est importante ; elle est appelée par le SDK dès que ce dernier s'est chargé. Pour réaliser toutes les manipulations sur le jeu, nous devons initialiser en précisant l'identifiant de l'application dans la propriété `appId`.

## L'utilisateur est-il connecté ?

Cette question est fondamentale pour les permissions, le partage et la sauvegarde de la progression. Si l'utilisateur est connecté, il peut jouer directement sans passer par la demande du mot de passe. Dans le cas contraire, nous l'invitons à se connecter avant de continuer.

Pour cela, nous utilisons deux scènes :

- une scène nommée `Login` pour rediriger le joueur vers la connexion à l'aide d'un bouton ;
- la scène principale du jeu (ou écran titre), nommée `Game`.

### Sélectionner la scène selon le statut

La sélection de la scène se fait dès que le `canvas` est prêt à l'emploi.

#### Choisir la scène selon le statut

```
var canvas = CE.defines("canvas");
ready(function() {
    FB.getLoginStatus(function(response) {
        if (response.status === 'connected') {
            var uid = response.authResponse.userID;
            canvas.Scene.call("Game");
        } else if (response.status === 'not_authorized') {
            // Ici, mettez une redirection vers la page d'autorisation
        } else {
            canvas.Scene.call("Login");
        }
    });
});
```

La méthode `getLoginStatus` récupère l'état de l'utilisateur qui peut prendre plusieurs valeurs.

Tableau 16-1. État de l'utilisateur

Condition	Valeur de l'état	Que fait-on ?
L'utilisateur est déjà connecté.	connected	Nous appellons la scène <code>Game</code> .
L'utilisateur est connecté mais n'a pas autorisé l'application.	not_authorized	Nous l'invitons à y remédier en le redirigeant vers la page d'autorisation (voir chapitre précédent).
L'utilisateur n'est pas connecté.	unknown	Nous appellons la scène <code>Login</code> .

**REMARQUE L'identifiant de l'utilisateur**

Dans le code proposé, nous récupérons l'identifiant de l'utilisateur. Celui-ci sert à retrouver les données dans notre propre base de données et manier l'utilisateur sur le serveur pour un jeu multijoueur.

## Scène pour demander une connexion

La scène se nomme `Login` et possède un bouton. Quand le joueur clique dessus, il est invité à saisir son nom d'utilisateur et son mot de passe.

### Scène pour la connexion

```
canvas.Scene.new({
    name: "Login",
    ready: function(stage) {
        var w=100, h=40;
        var button = this.createElement(w, h);

        // rectangle rouge pour le test
        button.fillStyle = "#ff0000";
        button.fillRect(0, 0, w, h);

        button.on("click", function() {
            FB.login ①(function(response) {
                if (response.authResponse) {
                    canvas.Scene.call("Game"); ②
                } else {
                    // Si connexion échouée
                }
            });
        });

        stage.append(button);
    }
});
```

Le bouton est un élément classique doté d'un événement « clic de la souris ». Nous utilisons la méthode `login` pour appeler la fenêtre de Facebook et effectuer la connexion ①. Après la saisie correcte, nous appelons la scène `Game` pour commencer le jeu ②.

La suite montrera l'utilisation des données du joueur dans la scène du jeu.

**ASTUCE Déclencher une fonction selon le changement d'état de l'utilisateur**

Vous pouvez attacher un événement à des écouteurs pour l'exécuter à leur appel. Il existe un événement nommé `auth.authResponseChange` pour le changement d'état :

```
FB.Event.subscribe('auth.authResponseChange', function(response) {  
    console.log(response.status);  
});
```

Ici, quand un état change, la fonction de rappel est déclenchée en indiquant cet état. Les valeurs sont identiques à la méthode `FB.getLoginStatus()`.

## Inviter des amis à jouer

Inviter un ami permet d'amorcer la « viralité » de votre jeu sur le réseau social. L'idée est d'envoyer une notification à des amis pour gagner de nouveaux joueurs.

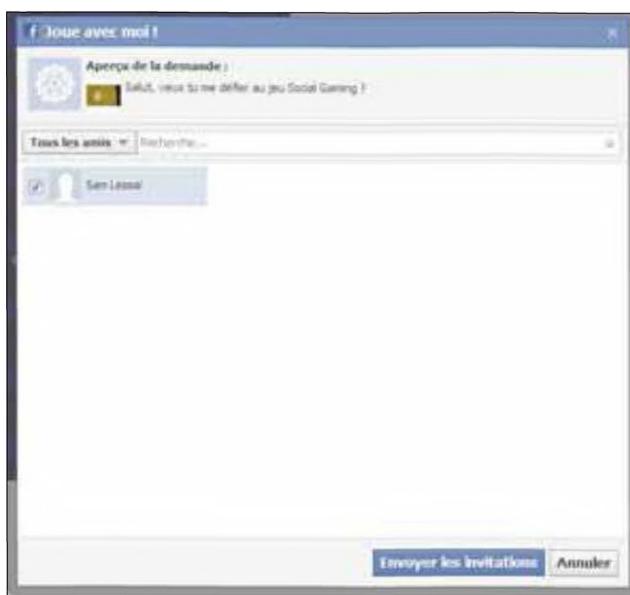
Facebook possède des boîtes de dialogue. C'est la méthode `ui` qui gère les interfaces.

### Interface sur Facebook

```
FB.ui({  
    method: 'apprequests',  
    title: 'Joue avec moi !',  
    message: 'Salut, veux-tu me défier au jeu Social Gaming?'  
, function(response) {  
    console.log(response);  
});
```

La propriété la plus importante est `method`. Elle définit le type de la boîte de dialogue. Ici, `apprequests` indique à Facebook d'afficher la liste des amis de l'utilisateur pour procéder aux invitations.

**Figure 16-1**  
Inviter des amis à jouer



Après l'envoi des invitations, un objet de la forme suivante est retourné :  
La clé `to` est un tableau comprenant tous les identifiants des amis invités.

**Figure 16-2**  
Objet retourné

```
* Object
  requestID: "2013433333333333"
  + to: Array[2]
    0: "4937433333333333"
    1: "4937433333333333"
  + requestID: 2
  + _ref: null
  + _index: 0
  + _isRef: false
```

## Supprimer la notification

Lorsque l'invitation est acceptée par l'ami du joueur, la notification n'est pas enlevée automatiquement. C'est au développeur d'effectuer cette étape. Un appel à la page `index.php` est lancé. En en-tête de cette page, supprimez la notification à l'aide du SDK PHP (voir le chapitre précédent pour son intégration) :

### Suppression de la notification

```
<?php
require 'src/facebook.php';

if(isset($_REQUEST['request_ids'])) {
  $requestIDs = explode(' ', $_REQUEST['request_ids']);
  foreach($requestIDs as $requestID) {
    try {
      $delete_success = $facebook->api('/' . $requestID, 'DELETE');
```

```

        } catch(FacebookAPIException $e) {
            error_log($e);
        }
    }
}

// code

?>

```

Toutes les notifications présentes sont parcourues à l'aide d'une boucle et supprimées avec l'API de Facebook.

## Afficher un score et le partager sur le mur de l'utilisateur

Pour défier ses amis, le joueur souhaiterait montrer son score. Facebook propose un système de score pour les applications de type « jeu ». Deux conditions sont requises :

- La permission `publish_actions` doit être affectée à l'application (voir chapitre précédent).
- Nous devons utiliser les jetons d'accès (*access token*) pour poster le score sur le mur en toute sécurité.

À la racine de votre jeu, créez un dossier `server` avec un fichier PHP nommé `savescore.php`. Ajoutez le code suivant :

### Partage du score

```

<?php
require 'src/facebook.php';

define('APP_ID', '386722101395435');
define('APP_SECRET', '5e2c6440583743a47b4e*****');
$score = $_REQUEST['score'];

$facebook = new Facebook(array(
    'appId'  => APP_ID,
    'secret' => APP_SECRET,
));

$user = $facebook->getUser();

$app_access_token = get_app_access_token(); ①
$facebook->setAccessToken($app_access_token); ④
$response = $facebook->api ⑤('/' . $user . '/scores', 'post', array(
    'score' => $score
));
print($response);

```

```

function get_app_access_token() {
    $token_url ③ = 'https://graph.facebook.com/oauth/access_token?'
        . 'client_id=' . APP_ID
        . '&client_secret=' . APP_SECRET
        . '&grant_type=client_credentials';

    $token_response =file_get_contents($token_url); ②
    $params = null;
    parse_str($token_response, $params);
    return $params['access_token'];
}
?>

```

Le début du code reprend la démarche énoncée dans le chapitre précédent : l'intégration du PHP SDK et la déclaration des constantes contenant identifiant et clé secrète de l'application.

La fonction `get_app_access_token`, que nous créons, récupère le jeton d'accès ② en ouvrant une page dont l'URL a été précédemment construite avec les constantes ③.

La méthode `setAccessToken` ④ affecte le jeton d'accès à l'instance `$facebook` pour enregistrer le score ensuite.

La méthode `api` envoie le score à Facebook pour l'utilisateur courant ⑤.

Le script PHP conçu, ajoutons l'appel Ajax dans le jeu :

#### Modèle pour l'envoi du score

```

Class.create("Game_Score", {
    send: function(score) {
        CE.ajax({
            type: 'POST',
            url: 'server/savescore.php?score=' + score,
            success: function(response) {
                if (response == '1') { // teste la valeur de
print($response)
➡ de PHP
                    // Envoi du score effectué
                }
            }
        });
    }
});

```

Le modèle `Game_Score` nous servira à envoyer le score. Une simple requête Ajax appelle le script PHP avec le score dans l'URL. Si la procédure est correcte, la valeur renvoyée vaudra 1 (valeur renournée de PHP).

Dans la méthode `ready`, initialisons la classe :

**Initialisation de la classe Game\_Score**

```
var game_score = Class.new("Game_Score");
```

Un simple bouton envoie un score :

**Envoi du score au clic sur l'élément**

```
var btn = this.createElement(50, 100);
// Code
btn.on("click", function() {
    game_score.send(1500); // Exemple
});
```

## Système de badges

Un badge est une récompense reçue lorsqu'un objectif précis est atteint (par exemple, badges Expert, Guerrier...).

Ainsi, un badge est gagné en fonction d'une condition et des règles du jeu établies. L'obtenir est secondaire et n'est pas indispensable dans la progression du jeu. Il permet surtout de partager fièrement avec ses amis un avancement ou un accomplissement.

Lorsque le badge est obtenu, nous devons le mentionner sur le mur de joueur. Ajoutez-le dans votre jeu en HTML5 :

**Boîte de dialogue pour le partage**

```
FB.ui({
    method: 'feed',
    caption: 'J\'ai obtenu le grade "Aventurier"',
    name: 'Jouez à Social Gaming avec votre ami !',
    picture: 'http://monsite.com/img.png',
    link: 'http://monsite.com/socialgaming/'
}, function(ret) {
    console.log(ret);
});
```

**REMARQUE console.log**

L'instruction `console.log` va rester dans le code pour vérifier le retour dans notre exemple.

**Tableau 16-2. Explication des propriétés**

Propriété	Explication
<code>method</code>	Valeur <code>feed</code> pour indiquer à Facebook que nous souhaitons poster un message sur le mur du joueur.
<code>caption</code>	Description du badge

name	Texte sur le lien, titre du badge
picture	L'image associée au badge
link	Lien vers le jeu ou une page explicative des badges

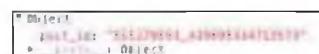
Le résultat est le suivant :

**Figure 16-3**  
Partager un avancement



Après la validation, l'identifiant du message est renvoyé :

**Figure 16-4**  
Objet envoyé après  
la validation du partage



## Récupérer des informations (amis, groupes, etc.) pour les utiliser dans le jeu

Les données de l'utilisateur sont récupérées avec l'API pour les afficher et les utiliser dans le jeu. Avec la méthode `api`, nous relevons :

- les données générales;
- la liste des amis;
- le groupe.

### Récupérer des données

```
FB.api(path, method, params, cb);
```

Tableau 16-3. Explication des paramètres

Nom	Type	Description
path	String	Chemin. La liste est présente dans le Graph API de Facebook.
method	String	Méthode GET (par défaut) ou POST
params	Object	Paramètres à envoyer
cb	Function	Fonction de rappel

**PRÉCISION Graph API de Facebook**

Graph API est un moyen simple pour le développeur de récupérer les données de l'utilisateur (amis, photos, événements, pages, etc.) et d'interagir avec elles.

Les paramètres `method` et `params` sont facultatifs. Pour restreindre l'acquisition des champs, ajoutez la propriété `fields` dans `params` en indiquant les champs souhaités, séparés par des virgules. Par exemple :

**Limiter les champs**

```
FB.api(path, {
  'fields': 'id,name'
}, function(response) {
});
```

**ASTUCE Tester en ligne**

Pour tester l'API avec des permissions spéciales directement en ligne, rendez-vous sur le lien suivant :

► <http://developers.facebook.com/tools/explorer>

## Données générales

Les données générales comprennent le nom, l'adresse électronique, l'identifiant, la langue, etc.

**Qui suis je?**

```
FB.api('/me', function(response) {
  console.log(response);
});
```

Le chemin `/me` renvoie un objet avec les différentes données.

**Figure 16-5**

## Les données de l'utilisateur

```
* Object
  email: "jean@cominelli.com"
  first_name: "Jean"
  gender: "male"
  id: "1000003333333434"
  last_name: "Leissé"
  link: "http://www.facebook.com/jean.leisse"
  locale: "fr_FR"
  name: "Jean Leisse"
  timezone: 1
  updated_time: "2012-08-19T17:45:17+0000"
  username: "jean.leisse5"
  verified: true
* __proto__: Object
```

**ALLER PLUS LOIN** Données générales de l'utilisateur

Une liste complète est proposée à l'adresse suivante :

► <http://developers.facebook.com/docs/reference/api/user/>

## Liste des groupes

Le joueur fait partie de plusieurs groupes. Pour les connaître, vérifiez que l'application a la permission `user_group`.

## Les groupes de l'utilisateur

```
FB.api('/me/groups', function(response) {  
    var data;  
    for (var i=0 ; i < response.data.length ; i++) {  
        data = response.data[i];  
        console.log(data.name, data.id);  
    }  
});
```

Le tableau `data` contient tous les groupes de l'utilisateur. Ici, `name` est le nom du groupe et `id` son identifiant.

**ALLER PLUS LOIN** Groupes de l'utilisateur

Une liste complète est proposée à l'adresse suivante :

► <http://developers.facebook.com/docs/reference/api/group/>

## Liste des amis

La liste des amis est beaucoup utilisée dans les jeux sociaux :

## La liste des amis

```
FB.api('/me/friends', function(response) {  
    var data;  
    for (var i=0 ; i < response.data.length ; i++) {  
        data = response.data[i];
```

```
    console.log(data.name, data.id);  
});
```

Deux propriétés sont renvoyées dans le tableau des amis : le nom et l'identifiant. Ce dernier est primordial ! Dans un jeu multijoueur, nous comparons cet identifiant avec celui de l'adversaire pour savoir s'il est un ami du joueur. Ainsi, dans le jeu, nous pouvons afficher l'avatar de l'ami (<https://graph.facebook.com/ID/picture>).

Enfin, si vous avez besoin de plus d'informations sur l'ami, ajoutez les commandes suivantes :

#### Informations sur un ami

```
FB.api('/' + ID, function(response) {  
    console.log(response);  
});
```

Les données renvoyées sont du même style que les données générales.



## Stratégie de monétisation

---

Votre jeu est terminé. Comment rentabiliser le fruit de votre travail ? Élaborez et déployez une stratégie de monétisation. Explication.

L'étude de marché, abordée dans le premier chapitre, montre que le *Social Gaming* est un élément indispensable dans le chiffre d'affaire des acteurs du Web. Pourtant, la plupart des jeux sont gratuits. Comment se rémunérer à travers ce type de jeu ? Nous devons établir une stratégie de monétisation et Facebook nous aide sur ce point.

## Monnaie virtuelle

### Pourquoi une monnaie virtuelle ?

Les jeux sont le plus souvent des *Free-To-Play*, c'est-à-dire que la totalité du jeu est gratuite mais que des services annexes ou des objets sont payants. Ces micro-paiements apportent des revenus conséquents lorsque le nombre de joueurs est important. Les petits ruisseaux font les grandes rivières !

Le jeu doit contenir une monnaie virtuelle. Il y a à cela plusieurs intérêts :

- appel simple à plus de 80 moyens de paiement dans plus de 50 pays ;
- gestion et ajustement des prix en fonction des facteurs économiques ;
- expérience de jeu plus simple pour le joueur.

#### DÉTAILS Modes de paiement

Les modes de paiement les plus populaires sur Facebook sont :

- les cartes de crédit : American Express, Discover, MasterCard, Visa ;
- PayPal ;
- numéro de téléphone mobile ;
- cartes cadeaux.

Les autres modes de paiement locaux sont affichés sur le lien :

▶ <http://www.facebook.com/help/?faq=203680236341574>

Par exemple, quand un objet est débloqué dans la suite logique du scénario, l'utilisateur peut l'acheter.

Figure 17-1

Objet débloqué à acheter



L'idée de laisser le choix au joueur en créant un catalogue favorise un achat potentiel.

**Figure 17-2**  
Choix d'un objet dans un catalogue



## Configuration et déploiement

Tout d'abord, activons le système des crédits. Pour cela, accémons à la page de configuration de l'application et cliquons sur *Crédits* dans le menu à gauche.

Dans *URL de notification*, entrez le lien vers la page PHP.

**Figure 17-3**  
Configuration des crédits

Créons le fichier PHP indiqué dans la configuration :

### Envoyer les informations sur un objet

```
<?php  
$app_secret = '5e2c6440583743a47b4e*****';
```

```
$request = parse_signed_request($_POST['signed_request'], $app_secret);

$request_type = $_POST['method'];

$response = '';

if ($request_type == 'payments_get_items') ❶{
    $order_info = json_decode($request['credits']['order_info'], true);

    $item_id = $order_info['item_id'];

    if ($item_id == '1a') {
        $item = array(
            'title' => 'Armure',
            'description' => 'Armure unique dans le jeu',
            'price' => 1,
            'image_url' => 'http://path/to/img.jpg',
        );
    }

    $response = array(
        'content' => array(
            0 => $item,
        ),
        'method' => $request_type,
    );
    $response = json_encode($response);
}
}

echo $response;

// Documentation sur ces méthodes à l'adresse :
// https://developers.facebook.com/docs/authentication/signed_request/
function parse_signed_request($signed_request, $secret) {
    list($encoded_sig, $payload) = explode('.', $signed_request, 2);

    $sig = base64_url_decode($encoded_sig);
    $data = json_decode(base64_url_decode($payload), true);

    if (strtoupper($data['algorithm']) != 'HMAC-SHA256') {
        error_log('Unknown algorithm. Expected HMAC-SHA256');
        return null;
    }

    $expected_sig = hash_hmac('sha256', $payload, $secret, $raw = true);
    if ($sig !== $expected_sig) {
        error_log('Bad Signed JSON signature!');
        return null;
}
```

```
    return $data;
}

function base64_url_decode($input) {
    return base64_decode(strtr($input, '-_', '+/'));
}
?>
```

Deux méthodes peuvent être reçues :

- `payments_get_items` ① : récupérer des informations pour faire un achat;
- `payments_status_update` : autres types (gain, contestation, remboursement de l'achat).

C'est la première requête qui nous intéresse.

#### DOCUMENTATION Signature

La méthode de création de la signature est documentée sur le lien suivant :

▶ [https://developers.facebook.com/docs/authentication/signed\\_request/](https://developers.facebook.com/docs/authentication/signed_request/)

Pour afficher la boîte de dialogue, utilisons la méthode `ui` :

#### Boîte de dialogue pour acheter un objet

```
FB.ui({
    method: 'pay',
    action: 'buy_item',
    order_info: {'item_id': '1a'},
    dev_purchase_params: {'oscif': true}
}, function(data) {

    if (data.order_id) {
        console.log("Succès : Order ID : " + data.order_id + " , Status : " +
        ↵ data.status);
    } else if (data['error_code']) {
        console.log("Echec : " + data.error_code + " , " + data.error_message);
    } else {
        console.log("Echec");
    }
});
```

Ce code peut être exécuté lorsque le joueur appuie sur un bouton. Il appellera le script PHP défini dans les configurations déclarées plus haut. Quatre propriétés sont présentes :

Tableau 17-1. Explication des propriétés

Nom	Type	Explication
method	String	Nom de la méthode pour cette action, soit pay.
action	String	Type de l'action. Les valeurs possibles sont buy_item, buy_credits, earn_credits, earn_currency. Nous voulons acheter un objet, donc nous utilisons la valeur buy_item.
order_info	Object	Envoi de données utilisables côté PHP.
dev_purchase_params	Object	Définit si la boîte de dialogue affiche la monnaie locale.

La fonction de rappel reçoit la réponse en JSON du script PHP. Si la propriété `order_id` existe, cela signifie que la transaction a bien été effectuée. Dans le cas contraire, s'ils existent, vous pouvez afficher le code et le message d'erreur.

Pensez bien à informer l'utilisateur en affichant l'état de la transaction. Le résultat est le suivant :

Figure 17-4

Le joueur achète des crédits pour obtenir l'objet.



A

## Rappels sur HTML5 Canvas

---

La balise `Canvas` était déjà disponible dans les versions précédentes de HTML, mais cette annexe se concentre sur celle de HTML5.

HTML5 dispose de plusieurs balises pour différentes applications web :

- création de WebTV avec la balise `<video>`;
- utilisation de la géolocalisation;
- utilisation de système de fichiers et stockage offline;
- création de messageries instantanées avec les WebSockets...

Nous allons nous concentrer sur des rappels ciblés sur la balise `<canvas>` pouvant être utilisés dans votre jeu.

## Initialiser et charger le canvas

Avant toute manipulation sur HTML5, nous avons besoin d'initialiser le contexte du `canvas`. Notre page HTML est épurée et intègre le fichier JavaScript :

### Simple page HTML5 contenant la balise `<canvas>` et le fichier JS à charger

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" charset="utf-8" src="main.js"></script>
  </head>
  <body>
    <canvas id="canvas" width="640px" height="480px">
      Votre navigateur Web ne prend pas en charge HTML5 Canvas.
    </canvas>
  </body>
</html>
```

La balise `<canvas>` possède un identifiant qui nous servira pour récupérer l'élément dans le fichier JavaScript. Dans le cas où nous sommes en présence d'un navigateur ne reconnaissant pas HTML5, nous le signalons entre les balises `<canvas>`.

Le fichier JavaScript, nommé `main.js`, se trouve à la racine du projet et contient le code suivant :

### Chargement et vérification du canvas

```
window.onload = function() {
  var el, ctx;
  el = document.getElementById('canvas');
  if (!el || !el.getContext) {
    return;
  }
  ctx = el.getContext('2d');
  // Suite du code
}
```

Lorsque la page est chargée, on récupère l'élément et on vérifie si la récupération du contexte est possible dans le cas où le navigateur de l'utilisateur ne permet pas de dessiner dans le `canvas`.

#### FRAMEWORK Déetecter la compatibilité avec Modernizr

Modernizr est une bibliothèque qui détecte les caractéristiques de HTML5 et CSS3 compatibles avec le navigateur de l'utilisateur. Téléchargez-la sur le lien suivant :

► <http://modernizr.com>

Insérez-la en en-tête de la page HTML5 :

```
<script type=>text/javascript</script> src=>modernizr.min.js</script>
```

Ensuite, procédez à la vérification :

```
if (Modernizr.canvas) {  
    // Le canvas est reconnu  
}  
else{  
    // le canvas n'est pas reconnu  
}
```

## Dessiner dans le canvas

Après l'initialisation, passons à l'étape dessin qui ouvre à nombre de possibilités : dessiner une ligne, un rectangle, des formes, intégrer des images, des textes et même des vidéos. Bien entendu, ces codes sont à insérer après la récupération du contexte.

### Les lignes

#### Une suite de lignes rouges

```
ctx.moveTo(10, 10);  
ctx.lineTo(100, 100);  
ctx.lineTo(170, 120);  
ctx.lineTo(240, 100);  
ctx.lineTo(330, 10);  
ctx.lineWidth = 2;  
ctx.strokeStyle = "#ff0000";  
ctx.stroke();
```

Nous plaçons le point de départ à (10,10) pixels et nous traçons une ligne jusqu'à (100,100), ensuite une autre jusqu'à (170,120), etc.

Nous appliquons une épaisseur de 2 pixels et une couleur rouge (#ff0000) que nous rendons visible avec la méthode `stroke`.

**ASTUCE Couleur transparente**

Au lieu de déclarer une couleur en hexadécimal pour `strokeStyle` ou `fillStyle`, il est possible d'appliquer le code RGBA (*Red, Green, Blue, Alpha*).

```
| ctx.strokeStyle = «rgba(255, 0, 0, 0.5)»;
```

La couleur sera rouge, mais semi-transparente. Alpha est compris entre 0 et 1 où 0 est la transparence totale.

## Les arcs

**Un arc**

```
| ctx.arc(100, 100, 90, 0.5 * Math.PI, 0, false);  
| ctx.strokeStyle = "#000";  
| ctx.stroke();
```

Pour dessiner un arc, nous utilisons la méthode `arc` avec les paramètres suivants :

**Paramètres de la méthode `arc()`**

```
| ctx.arc(x, y, radius, startingAngle, endingAngle, antiClockwise);
```

- `x` : position X du centre de l'arc;
- `y` : position Y du centre de l'arc;
- `radius` : taille du rayon en pixels;
- `startingAngle` : angle en radians pour commencer à dessiner la ligne courbée sur la circonférence du cercle;
- `endingAngle` : angle en radians pour terminer la ligne;
- `antiClockwise` : si sa valeur vaut `true`, l'arc sera dessiné dans le sens contraire des aiguilles d'une montre.

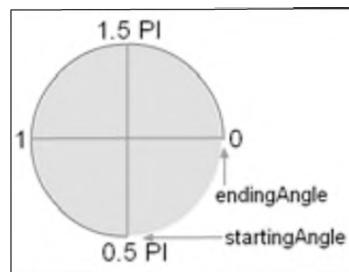
Ici, nous avons dessiné un cercle presque entier en plaçant le centre aux positions (100,100).

**ASTUCE Dessiner un cercle complet**

```
| ctx.arc(x, y, radius, 0, 2 * Math.PI, false);
```

**Figure 18-1**

Représentation trigonométrique



## Chemins

Un chemin est une combinaison de lignes, d'arcs, de courbes quadratiques et de Bézier. Cela signifie que les méthodes suivantes doivent être utilisées :

- `lineTo`;
- `arcTo`;
- `quadraticCurveTo`;
- `bezierCurveTo`.

Pour commencer un chemin, utilisez la méthode `beginPath`.

## Les formes

### Rectangle

#### Dessiner un rectangle rouge

```
ctx.rect(0, 0, 100, 100);
ctx.fillStyle = "red";
ctx.fill();
```

#### Paramètres de la méthode `rect()`

```
ctx.rect(x, y, width, height);
```

Dessiner un rectangle est un jeu d'enfant, utilisez la méthode `rect` avec quatre paramètres :

- `x` : position X du point supérieur gauche en pixels ;
- `y` : position Y du point supérieur gauche en pixels ;
- `width` : largeur en pixels ;
- `height` : hauteur en pixels.

Nous affectons une couleur de remplissage avec la méthode `fillStyle` et nous l'appliquons au rectangle avec `fill`.

## Recadrer

Nous pouvons recadrer une forme dans une autre avec la méthode `clip`. Cela signifie que seul ce qui est à l'intérieur de la forme de recadrage sera affiché; les parties extérieures seront cachées.

### Recadrer un cercle dans un rectangle

```
var x = 50, y = 50, w = 100, h = 70;
ctx.beginPath();
ctx.rect(x, y, w, h);
ctx.clip();

ctx.beginPath();
ctx.arc(100, 60, 50, 2 * Math.PI, 0, false);
ctx.fillStyle = "red";
ctx.fill();
ctx.closePath();

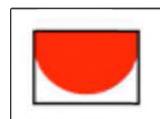
ctx.beginPath();
ctx.rect(x, y, w, h);
ctx.strokeStyle = "black";
ctx.lineWidth = 5;
ctx.stroke();
ctx.closePath();
```

Dans ce code, nous procédons en trois étapes :

- 1 Nous définissons un rectangle de taille  $100 \times 70$  px. Toutes les formes qui suivront seront recadrées selon ce rectangle.
- 2 Nous dessinons un cercle dont la partie supérieure dépasse du rectangle et sera donc cachée.
- 3 Nous dessinons le rectangle pour visualiser son emplacement.

**Figure 18-2**

Cercle recadré dans le rectangle



Pour continuer à dessiner tout en ignorant le recadrage ensuite, il faut procéder de la manière suivante :

- 1 Avant de définir la forme de recadrage (en première ligne de code dans l'exemple précédent) :

### Sauvegarder l'état actuel du canvas

```
| ctx.save();
```

2 Avant de dessiner des formes en dehors du cadre :

#### Restaurer l'état

```
ctx.restore();
```

## Les dégradés

### Dégradé linéaire

#### Appliquer un dégradé sur un rectangle

```
var gradient = ctx.createLinearGradient(50, 0, 50, 100);
gradient.addColorStop(0, "#000");
gradient.addColorStop(1, "#fff");
ctx.rect(0, 0, 100, 100);
ctx.fillStyle = gradient;
ctx.fill();
```

#### Paramètres de createLinearGradient()

```
ctx.createLinearGradient(x0, y0, x1, y1);
```

Le tracé du dégradé commence au point (x0, y0) et se termine en (x1, y1).

**Figure 18-3**

Direction du tracé du dégradé noir vers le blanc



La première couleur est noire (#000) et s'éclaircit vers le blanc (#fff). Nous appliquons les couleurs avec `addColorStop`. Vous pouvez ajouter cette méthode autant de fois que vous le souhaitez, avec différentes couleurs, sur une ligne imaginaire comprise entre 0 et 1 afin de faire des variations de couleurs.

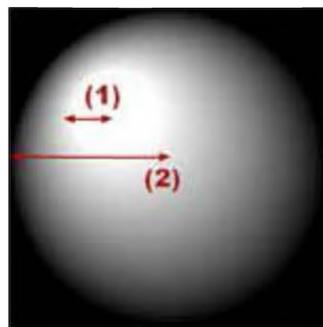
### Dégradé radial

Pour dessiner un dégradé radial, nous utilisons la méthode `createRadialGradient`.

#### Appliquer un dégradé radial sur un rectangle

```
var gradient = ctx.createRadialGradient(100, 100, 40, 150, 150, 150);
gradient.addColorStop(0, "#fff");
gradient.addColorStop(1, "#000");
ctx.rect(0, 0, 300, 300);
ctx.fillStyle = gradient;
ctx.fill();
```

**Figure 18-4**  
Dégradé radial



Le dégradé superpose deux cercles. L'épicentre éclaire et le contour assombrit. Les couleurs sont définies avec `addColorStop` comme pour les dégradés linéaires.

Les trois premiers paramètres de `createRadialGradient` permettent de définir les positions et le rayon de l'épicentre. Les trois derniers sont pour le second cercle.

#### Paramètres

```
ctx.createRadialGradient(x0, y0, r0, x1, y1, r1);
```

Sur la figure par exemple :

- (1) `x0` et `y0` situent le centre du premier cercle, de rayon `r0` en pixels.
- (2) `x1` et `y1` situent le centre du second cercle, de rayon `r1` en pixels.

## Afficher un texte

### Afficher un texte

```
ctx.font = "20px Arial";
ctx.fillStyle = "#000";
ctx.fillText("Un texte", 0, 100);
```

Afficher un texte est très simple :

- `font` : taille en pixels ou en points, ainsi que la police d'écriture ;
- `fillStyle` : couleur du texte ;
- `fillText` : chaîne de caractères à afficher aux positions X et Y.

Remarquez qu'on pourrait être tenté de placer le texte aux positions (0,0) afin de l'afficher au coin supérieur gauche. Cependant, la position Y correspond à la ligne située en bas du texte. Cela signifie que votre texte sera à l'extérieur du `canvas` et donc caché. La solution consiste :

- soit à positionner soi-même le texte sur la position Y ;
- soit à ajuster la ligne du texte avec la propriété `textBaseline`.

Dans ce dernier cas, le code sera le suivant :

#### Ajuster la ligne du texte

```
ctx.textBaseline = "top";
```

Cette propriété peut en fait prendre différentes valeurs :

- `top` : ligne en haut;
- `middle` : ligne au milieu;
- `bottom` : ligne en bas;
- `ideographic` : ligne en bas qui ignore les descendantes. Valeur par défaut.

#### VOCABULAIRE Descendante

Une descendante, en typographie, est la partie d'une lettre descendant sous la ligne de base des autres lettres. Par exemple, « g » est une descendante.

La propriété `textAlign` sert pour aligner horizontalement le texte :

#### Centrer la ligne du texte

```
ctx.TextAlign = "center";
```

Cette propriété peut prendre trois valeurs :

- `left` : alignement sur la gauche;
- `right` : alignement sur la droite;
- `center` : centrer.

## Couleurs

Si nous souhaitons seulement remplir le texte d'une couleur, la propriété `fillStyle` suffit. L'exemple plus haut montre un texte rempli de la couleur noire.

En revanche, si nous voulons appliquer une couleur seulement à la bordure, c'est la propriété `strokeStyle` qui rentre en jeu :

#### Couleur rouge sur les bordures du texte

```
ctx.strokeStyle = "red";
ctx.strokeText("Un texte", 0, 100);
```

## Les ombres

Pour ajouter une ombre sur une forme, utilisez les propriétés :

- `shadowColor` : couleur de l'ombre en hexadécimal;

- shadowBlur : intensité du flou de l'ombre;
- shadowOffsetX : décalage de l'ombre vers la droite;
- shadowOffsetY : décalage de l'ombre vers le bas.

#### Ombre noire sur un carré rouge

```
ctx.rect(50, 50, 100, 100);
ctx.fillStyle = "#ff0000";
ctx.shadowColor = "#000000";
ctx.shadowBlur = 10;
ctx.shadowOffsetX = 10;
ctx.shadowOffsetY = 10;
ctx.fill();
```

Pour un décalage de l'ombre sur la gauche ou vers le haut, mettez des valeurs négatives aux propriétés shadowOffsetX et shadowOffsetY. Si leur valeur vaut 0, une ombre uniformisée sera affichée autour de la forme.

## Composite

Il est possible de faire des opérations entre une image source A, qui peut être une forme ou une image, et une image de destination B. Pour cela, avant de dessiner une forme, nous utilisons la propriété globalCompositeOperation.

#### L'intersection des images A et B est transparente

```
ctx.globalCompositeOperation = "xor";
```

**Figure 18-5**  
Compositions

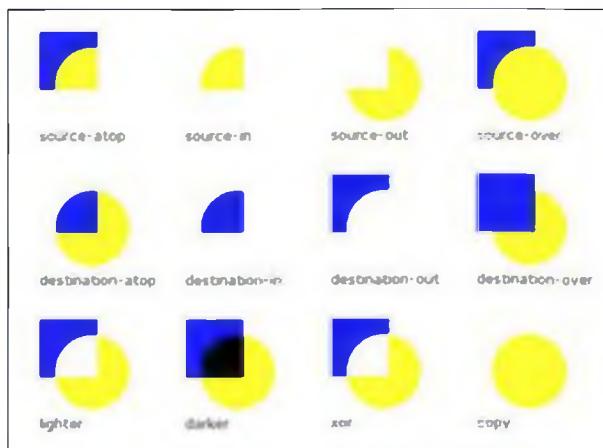


Tableau 18-1. Explication des valeurs de globalCompositeOperation

Propriété	Explication
source-atop	Rogner la partie extérieure de l'image A tout en la gardant.
source-in	Rogner la partie extérieure de l'image A et celle ci devient transparente.
source-out	Le contenu de l'image B est transparent.
source-over	L'image A se superpose à l'image B.
destination-atop	Rogner la partie extérieure de l'image B tout en la gardant.
destination-in	Rogner la partie extérieure de l'image B et celle ci devient transparente.
destination-out	Le contenu de l'image A est transparent.
destination-over	L'image B se superpose à l'image A.
lighter	L'intersection des images A et B est éclaircie.
darker	L'intersection des images A et B est assombrie.
xor	L'intersection des images A et B est transparente.
copy	Affichage seulement de l'image source A.

Pour changer la transparence de la globalité de l'image ou d'une forme, utilisez la propriété `globalAlpha` variant entre 0 (transparent) et 1 (opaque).

#### Valeur de la transparence

```
ctx.globalAlpha = 0.7;
```

## Insérer des images

Insérer des images dans le `canvas` est très courant dans les jeux. L'insertion se déroule en deux étapes : chargement de l'image, puis dessin de l'image chargée.

#### Chargement et affichage de l'image

```
var img = new Image();
img.src = "flower.png";
img.onload = function(){
    ctx.drawImage(this, 0, 0);      // objet Image à dessiner, positions X et Y
};
```

- Nous créons un instance de la classe `Image`.
- Nous lui affectons le chemin de l'image avec la propriété `src`.
- Nous lui indiquons la fonction à appeler lorsque l'image est chargée.
- Dans cette fonction, nous utilisons la méthode `drawImage` pour dessiner l'image en question.

`drawImage` ne fait pas qu'insérer une image sur le `canvas`. Nous pouvons également la redimensionner, la recadrer ou la découper.

Il est possible d'ajouter d'autres paramètres pour avoir plus de fonctionnalités.

#### Astuces Insérer un canvas au lieu d'une image

Remarquez qu'il est possible d'insérer un élément de type `Canvas` au lieu de l'image. Si le `canvas` existe déjà, nous pouvons le récupérer selon son identifiant avec `document.getElementById`, sinon nous le créons dynamiquement :

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

## Redimensionner

### Paramètres pour redimensionner l'image

```
ctx.drawImage(image, dx, dy, dw, dh);
```

Les deux derniers paramètres indiquent la largeur et la hauteur.

Ainsi, pour que notre dessin soit deux fois plus petit, nous divisons par deux la taille de l'image :

### Diviser par 2 la taille de l'image

```
ctx.drawImage(this, 0, 0, this.width/2, this.height/2);
```

## Couper

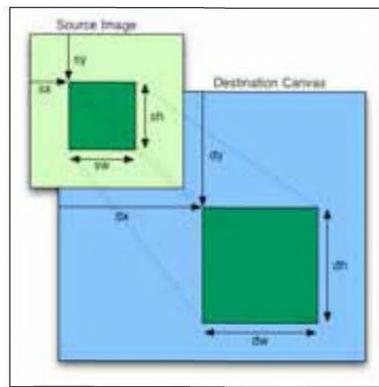
### Paramètres pour redimensionner l'image

```
ctx.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh);
```

Les paramètres `sx`, `sy`, `sw` et `sh` concernent l'image source. Cette formulation permet d'en prendre une partie et de l'insérer sur le `canvas` avec les paramètres `dx`, `dy`, `dw`, `dh`.

**Figure 18-6**

Schéma du fonctionnement de `drawImage()` pour couper une partie  
(source : W3C)



- L'illustration indique la largeur et la hauteur (`sw` et `sh` respectivement) de la partie à couper aux positions X et Y de l'image source (`sx` et `sy` respectivement) et son placement sur le canvas aux positions X et Y (`dx` et `dy` respectivement) avec une nouvelle taille (`dw` et `dh`). Si vous ne voulez pas changer sa taille, les paramètres `sw` et `sh` sont égaux à `dw` et `dh`.
- Voici un exemple. Nous avons l'image suivante :

**Figure 18-7**

Personnage à découper



Nous souhaitons ne prendre que le personnage du haut.

#### Prendre la partie supérieure de l'image

```
ctx.drawImage(this, 0, 0, this.width, this.height/2, 0, 0, this.width, this.height/2);
```

Nous gardons la largeur mais nous ne prenons que la moitié de l'image.

## Répéter l'image en fond

Après avoir chargé l'image, vous pouvez aussi la répéter en fond, en procédant de manière similaire au remplissage d'une forme par une couleur.

#### Remplir le fond d'un motif

```
var image = new Image();
image.src = "flower.png";
image.onload = function(){
    var pattern = ctx.createPattern(this, "repeat");
    ctx.rect(0, 0, canvas.width, canvas.height);
```

```

    ctx.fillStyle = pattern;
    ctx.fill();
}

```

Nous utilisons la méthode `createPattern` pour répéter l'image sur le rectangle créé. Le deuxième paramètre indique la répétition sur l'abscisse, l'ordonnée ou les deux :

- `repeat-x` : répétition du motif sur l'abscisse ;
- `repeat-y` : répétition du motif sur l'ordonnée ;
- `repeat` : répétition sur l'abscisse et l'ordonnée (par défaut, si le paramètre n'est pas renseigné) ;
- `no-repeat` : pas de répétition (cela revient à insérer l'image seule).

## Transformation

### Translation

La translation consiste à déplacer l'image ou la forme en se référant à un point de translation de coordonnées (a,b). Ce dernier se trouve par défaut aux positions (0,0) et les nouvelles coordonnées de l'image déplacée se calculent ainsi :

```

x' = x + a
y' = y + b

```

#### Translation de 50 pixels

```
ctx.translate(50, 50);
```

Par exemple, en mettant un rectangle aux positions (20,20), cette translation déplacera en réalité le rectangle aux positions (70,70).

### Rotation

Pour effectuer une rotation, il faut appliquer la méthode `rotate` au contexte :

#### Rotation de 50°

```
ctx.rotate(50*Math.PI/180);
```

#### RAPPEL La conversion des degrés aux radians

La formule mathématique est la suivante :

```
rad = deg * PI / 180
```

Notre rotation en radians est faite par rapport au point de translation. Ainsi, un rectangle ne tournera pas de 50 degrés sur lui-même mais par rapport au point placé à (0,0) par défaut.

#### Placement du point de transformation au milieu et rotation d'un rectangle

```
ctx.translate(el.width/2, el.height/2);
ctx.rotate(50*Math.PI/180);
ctx.fillStyle = "red";
ctx.fillRect(-50, -50, 100, 100);
```

Dans cet exemple, nous plaçons le point de translation au milieu du `canvas` et du rectangle et nous effectuons une rotation de 50 degrés. Le rectangle fera une rotation sur lui-même. Remarquez que le rectangle a été positionné de façon à placer le point en son milieu; pour cela, nous avons décalé sur la gauche et la hauteur la moitié de la taille du rectangle.

## Redimensionnement

C'est avec la méthode `scale` qu'on applique le redimensionnement :

#### Taille divisée par deux

```
ctx.scale(0.5, 0.5);
```

Cette méthode accepte deux paramètres :

- 1 `scaleX` : valeur comprise entre 0 et 1 pour le redimensionnement horizontal;
- 2 `scaleY` : valeur comprise entre 0 et 1 pour le redimensionnement vertical.

Logiquement, en donnant la valeur 0.5, on divise la taille par 2.

#### ASTUCE Effectuer un effet miroir

Affecter une valeur négative revient à réaliser un effet miroir. Par exemple :

```
ctx.scale(-1, 1);      // miroir horizontal
ctx.scale(1, -1);      // miroir vertical
```

## Transformation personnalisée

Pour appliquer une transformation personnalisée, utilisez la méthode `setTransform(a, b, c, d, e, f)` qui prend six paramètres représentant une matrice de dimension (3,3) :

**Figure 18-8**  
Paramètres de la matrice

a	c	e
b	d	f
0	0	1

- a : redimensionnement sur l'abscisse. Valeur 1 par défaut. En mettant une valeur 2, par exemple, la largeur de l'élément est doublée;
- b : inclinaison sur l'axe Y;
- c : inclinaison sur l'axe X;
- d : redimensionnent sur l'ordonnée;
- e : translation sur l'abscisse;
- f : translation sur l'ordonnée.

Prenons un exemple :

#### Inclinaison de 20 degrés et translation de 10 px

```
var a = 10 * Math.PI / 180; // degrés
ctx.setTransform(1, 0, Math.tan(a), 1, 10, 10);
ctx.fillStyle = "blue";
ctx.fillRect(0, 0, 100, 100);
```

#### Astuce Effectuer une rotation

Nous pouvons faire pivoter l'élément de 45 degrés avec la formule suivante :

```
var a = 45 * Math.PI / 180; // degrés
ctx.setTransform(Math.cos(a), Math.sin(a), -Math.sin(a), Math.cos(a), 0, 0);
```

## Manipulation des pixels

Sur le canvas, il est possible de récupérer les pixels afin de faire des traitements particuliers, appliquer des filtres ou transformer une image (la mettre en noir et blanc, par exemple).

#### Récupérer les pixels d'un canvas

```
var imgData = ctx.getImageData(0, 0, 100, 100);
```

Ici, c'est sur une zone de 100 pixels de largeur et de hauteur et à partir des positions (0,0) que les pixels sont récupérés. Nous aurons ainsi une matrice de 100\*100 entrées.

`getImageData` renvoie objet `ImageData` avec la taille et le tableau de pixels. Pour lire chaque pixel, nous faisons une boucle sur ce tableau.

### Diminuer l'opacité des pixels

```
var pixels = imgData.data, i;  
for (i=0; i < pixels.width * pixels.height * 4 ; i+=4) {  
    pixels[i+3] -= 100;  
}
```

Le tableau constitue la propriété `data` de l'objet `ImageData`. Nous parcourons le tableau de 4 en 4 pour prendre les couleurs du pixel. Ainsi, nous avons :

- pour `i=0` : valeur pour la couleur rouge de 0 à 255 pour le premier pixel;
- pour `i=1` : valeur pour la couleur verte de 0 à 255 pour le premier pixel;
- pour `i=2` : valeur pour la couleur bleue de 0 à 255 pour le premier pixel;
- pour `i=3` : valeur pour l'opacité (alpha) de 0 à 255 pour le premier pixel;
- pour `i=4` : valeur pour la couleur rouge de 0 à 255 pour le second pixel;
- pour `i=5` : valeur pour la couleur verte de 0 à 255 pour le second pixel;
- ...

Lorsque les modifications ont été effectuées, nous mettons les pixels sur le `canvas` aux positions X et Y définies avec la méthode `putImageData`.

### Ajouter les nouveaux pixels sur le canvas

```
ctx.putImageData(imgData, 10,30);
```

#### PERFORMANCE Utilisez une variable intermédiaire pour lire le tableau

Sur notre code, nous aurions très bien pu écrire ceci :

```
for (i=0; i < imgData.data.width * imgData.data.height * 4 ; i+=4) {  
    imgData.data[i+3] -= 100;  
}
```

Cependant, changer directement le tableau dans l'objet `ImageData` pose un problème de performance pour le traitement de l'image. Nous passons donc par une variable intermédiaire. Remarquez toutefois que cela change les entrées du tableau de l'objet `ImageData`. C'est pour cette raison que nous affectons la variable `imgData` et non `pixels` dans la méthode `putImageData`.



# B

## Frameworks JavaScript

---

Plusieurs frameworks ont été développés afin de faciliter et accélérer la conception de jeux. Remarquez cependant que ces frameworks ne séparent pas toujours l'affichage et les calculs, empêchant la mise en place de jeux multijoueurs.

Nous allons présenter deux frameworks :

- Easel.js : méthodes bas niveau pour faire des jeux ou applications avec HTML5 Canvas;
- RPG JS : bibliothèque très spécifique à la conception de jeux de rôle.

## Easel.js

Easel.js est une bibliothèque reprenant le nom des méthodes de Flash pour le `canvas`.

### Installer

En en-tête de votre page, insérez la bibliothèque téléchargée sur le site [createjs.com](http://createjs.com) ou mettez le lien direct vers le code source :

#### Insertion de la bibliothèque Easel.js

```
<script src="http://code.createjs.com/easeljs-X.Y.Z.min.js"></script>
```

Remplacez les lettres X, Y, et Z par la version courante.

### Premiers pas

Pour dessiner sur le `canvas`, il faut créer un objet `Stage`, lui ajouter les éléments et le mettre à jour :

#### Créer un objet Stage

```
var canvas = document.getElementById('canvas_1');
var stage = new Stage(canvas);
stage.update();
```

### Ajouter des formes

Pour dessiner des formes, nous devons utiliser un objet `Graphics` pour appliquer les méthodes de dessin et l'affecter à un objet `Shape` pour placer le dessin sur le `canvas` :

#### Dessiner un cercle rouge

```
var stage = new Stage(canvas);
var g = new Graphics();
g.beginFill(Graphics.getRGB(255,0,0));
g.drawCircle(10, 10, 50);
var shape = new Shape(g);
shape.x = 50;
shape.y = 75;
```

```
stage.addChild(shape);
stage.update();
```

Ici, nous dessinons un cercle d'un rayon de 50 pixels et de couleur rouge (récupérée avec la méthode `getRGB`).

Lorsque la forme est dessinée, nous l'affectons dans le paramètre du constructeur de la classe `Shape` pour le positionner sur le `canvas`.

Remarquez que la forme est un enfant du `canvas` principal.

#### À RETENIR Un enfant d'un élément

Comme pour la manipulation du DOM, un élément ajouté comme enfant avec la méthode `addChild` est rattaché à l'élément parent. Ainsi, si vous bougez, déformez, changez l'opacité ou autres du parent, les enfants prendront ses paramètres.

## Appeler le `canvas` en boucle

Pour les animations, la bibliothèque dispose d'une classe statique nommée `Ticker`. Chaque écouteur affecté sera appelé à un intervalle de temps régulier.

#### Appel de la fonction `tick()` en boucle

```
window.onload = function() {
    var canvas = document.getElementById('canvas_1');
    var stage = new Stage(canvas);
    Ticker.addListener(tick);
    stage.update();
}
function tick() {
    // Code appelé en boucle
}
```

Le paramètre de la méthode `addListener` est donc une fonction appelée en boucle.

## Afficher un texte

Pour afficher un texte, il faut utiliser l'objet `Text` :

#### Afficher un texte

```
var text = new Text ("Text", "12px bold Arial ", "#000000");
stage.addChild(text);
```

Le constructeur dispose de trois paramètres :

- la chaîne de caractères pour le texte ;
- le style dans un format CSS ;

- la couleur en hexadécimal.

La classe `Text` hérite de `DisplayObject`. Vous pouvez donc gérer son positionnement, sa rotation, etc. Il faut ensuite l'ajouter au `canvas` avec la méthode `addChild()`.

## Ajouter des conteneurs

Un conteneur est un objet héritant de la classe `DisplayObject` incluant divers éléments, aussi bien des formes que d'autres conteneurs.

### Déplacement de deux textes vers la droite à l'aide un conteneur

```
// Variables globales
var cont, stage;
window.onload = function() {
    // Déclaration des variables
    var canvas = document.getElementById('canvas_1'),
        text_y = 100,
        text1 = new Text ("Text 1", "Arial", "#ff0000"),
        text2 = new Text ("Text 2", "Arial", "#00ff00");
    stage = new Stage(canvas);
    cont = new Container();

    // Positionnement des textes
    text1.x = 100;
    text2.x = 200;
    text1.y = text2.y = text_y;

    // Ajout dans le conteneur et sur le canvas
    cont.addChild(text1);
    cont.addChild(text2);
    stage.addChild(cont);

    Ticker.addListener(tick);
}

function tick() {
    // Déplacement de 5 px vers la droite jusqu'à 600 px
    if (cont.x <= 600) {
        cont.x += 5;
        stage.update();
    }
}
```

Nous positionnons deux textes (rouge et vert) dans un conteneur. Les textes sont ajoutés avec la méthode `addChild()`.

Dans la fonction appelée en boucle, le conteneur se déplace vers la droite jusqu'à ce que sa position X soit égale à 600. Les deux textes sont bougés puisqu'ils appartiennent au conteneur.

## Insérer et animer une image

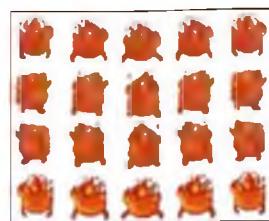
Pour insérer une image, il faut utiliser la classe `Bitmap` avec le lien de l'image ou un objet `Image` :

### Insérer une image

```
var bitmap = new Bitmap("images/spider.png");
stage.addChild(bitmap);
```

Pour une animation, vous devez disposer d'une image avec les différentes poses de l'animation :

**Figure 19-1**  
Image utilisée  
pour l'animation



### Animation

```
// Variable globale
var stage;
window.onload = function() {
    // Déclaration des variables
    var canvas = document.getElementById('canvas_1'),
        sheet, bitmap;
    stage = new Stage(canvas);

    // Chargement de l'image
    var img = new Image();
    img.src = "images/monstre.png";
    img.onload = function() {
        // Paramétrer le Sprite
        sheet = new SpriteSheet({
            images: [this],
            frames: {width: 250/5, height: 200/4},
            animations: {run:[0,4]}
        });
        bitmap = new BitmapAnimation(sheet);
        bitmap.gotoAndPlay("run");
        stage.addChild(bitmap);
    }
    Ticker.addListener(tick);
}

function tick() {
```

```

    stage.update();
}

```

Nous instancions une variable de la classe `Image` que nous insérons dans les paramètres du constructeur de la classe `SpriteSheet`. Cette dernière permet de découper l'image et de définir les animations. Cela se décompose en trois parties :

- 1 `images` : l'image à charger ou déclarée préalablement (notre cas);
- 2 `frames` : la largeur et la hauteur d'une animation. Notre image fait 250 px de largeur et 200 px de hauteur. Nous divisons donc la taille par le nombre de frames de notre image.
- 3 `animations` : le nom des animations avec les poses à lire en boucle. Ici, notre animation se nomme `run` avec un tableau comme valeur. Le tableau se décompose de la manière suivante :

```
[début, fin, animation suivante, fréquence]
```

Les deux dernières valeurs peuvent être omises. Dans notre exemple, on va lire la première pose (à partir de 0) jusqu'à la cinquième.

Si vous avez une autre animation, indiquez son nom dans la troisième entrée du tableau. Enfin, la fréquence permet d'accélérer ou de ralentir l'animation. Plus la valeur est grande, plus l'animation sera lente.

La variable de type `SpriteSheet` est le paramètre du constructeur de la classe `BitmapAnimation`. En fait, nous insérons l'image, mais cette fois animée.

Remarquez qu'on joue directement l'animation `run` avec `gotoAndPlay`. Pour ne jouer l'animation qu'une seule fois, vous avez la méthode `gotoAndStop` à votre disposition.

Pour voir l'animation se dérouler, il faut mettre à jour le `canvas` dans la fonction appelée en boucle.

## RPG JS : créez des jeux de rôle

RPG JS est basé sur Easel.js. Il implémente les mouvements, le défilement de l'écran, les animations, les actions, etc.

### Premiers pas

Dans un premier temps, téléchargez la bibliothèque sur [rpgjs.com](http://rpgjs.com) et insérez-la en en-tête de votre page :

#### Insertion de la bibliothèque RPG JS

```
<script src="rpg-beta.min.js"></script>
```

Nous allons ensuite charger la carte :

#### Chargement d'une carte

```
var rpg;
RPGJS.load(function() {
    rpg = new Rpg("canvas_rpg");
    rpg.loadMap('MyMap', {
        tileset: 'grassland.png',
        player: {
            x: 11,
            y: 8,
            filename: 'hero.png'
        }
    }, function () {
        rpg.setScreenIn("Player");
    });
});
```

Le constructeur de la classe `Rpg` a pour paramètre l'identifiant de la balise `<canvas>`. Nous chargeons la carte avec la méthode `loadMap` et trois paramètres :

- le nom du fichier JSON sans l'extension se trouvant dans le dossier `Data/Maps` (structure de la carte);
- les paramètres de la carte avec l'ensemble des carreaux positionnés dans le dossier `Graphics/Tilesets`;
- le placement du joueur et son apparence placés dans `Graphics/Characters`.

Lorsque la carte est chargée – fichier JSON et images chargés – nous pouvons utiliser l'API de RPG JS pour appliquer les fonctionnalités des RPG. Dans un premier temps, on fixe la caméra sur le joueur pour le déplacement de la carte.

#### Structure de la carte

```
{"map": [[[384,null,null],[384,null,null], [384,null,null]]] "proprietes":
  {"384": [0, 15]}}
```

Le fichier JSON est composé de deux paramètres :

- `map` : tableaux à trois dimensions, position X, position Y et une notion de couches de superposition (plus l'index de l'élément du tableau est élevé, plus le carreau sera en premier plan. La valeur de l'élément représente l'identifiant du carreau. Celui-ci doit commencer à partir de 384 car les identifiants en-dessous sont réservés au carreaux auto-créés);
- `proprietes` : objet où la clé est l'identifiant du carreau utilisé dans la carte avec un tableau à deux entrées comme valeur.

## Transférer le joueur sur une autre carte

Pour transférer le joueur sur une autre carte, il suffit d'utiliser la méthode `prepareMap` avec les mêmes paramètres que `loadMap`. Il faut omettre le paramètre `player` qui a été déjà défini dans `loadMap`.

### Préparation d'une carte

```
rpg.loadMap('MyMap', {
    tileset: 'grassland.png',
    player: {
        x: 11,
        y: 8,
        filename: 'hero.png',
        transfert: [
            {x: 0, y: 20, map: 'otherMap', x_final: 35, y_final: 20}
        ]
    }
}, function () {
    rpg.setScreenIn("Player");
});
rpg.prepareMap('otherMap', {
    tileset : 'grassland.png'
    transfert: [
        {x: 35, y: 20, map: 'myMap', x_final: 0, y_final: 20}
    ]
}, function() {
});
```

Pour transférer le joueur de la carte `myMap` vers `otherMap`, il faut indiquer les positions finales X et Y sur la seconde, où le joueur sera transféré à partir des positions de la première carte.

## Créer un événement

Un événement permet de faire vivre votre jeu. Cela peut être un personnage non joueur, une pierre à déplacer, un coffre à ouvrir, etc. Cet événement se place ensuite sur la carte et exécute des commandes selon des conditions de déclenchement.

- 1 Créez un fichier JSON dans le dossier `Data/Events/[Nom de la carte]` de votre jeu (par exemple : `EV001.json`) ;
- 2 Placez-y un tableau à deux entrées. Le premier élément est un objet ayant les paramètres généraux de l'événement, principalement ses positions, son nom et son identifiant. Le deuxième élément est un tableau de pages de l'événement.

### Configuration de l'événement – Paramètres globaux

```
[  
  {  
    "x": 16,  
    "y": 7,  
    "id": "1",  
    "name": "EV001"  
  },  
  [  
  ]  
]
```

La position X de l'événement est 16 et sa position Y est 7. Son nom ou son identifiant va permettre de les retrouver pour les manipuler avec le code (avec la méthode `getEventById` par exemple). Notez qu'un identifiant sera donné aléatoirement à l'événement si vous ne le précisez pas.

L'événement peut avoir plusieurs pages. Il exécutera seulement la dernière page si la condition précisée est remplie. Prenons un exemple :

### Configuration de l'événement – Condition de déclenchement

```
[  
  {  
  },  
  {  
    "conditions": {"self_switch": "A"},  
  }  
]
```

Admettons que l'interrupteur local A soit désactivé. Normalement, c'est la page 2 qui doit s'exécuter. Cependant, une condition a été affectée à cette page 2 : l'interrupteur local A doit être activé. Puisque ce n'est pas le cas, c'est la page 1 qui est exécutée. Remarquez que si vous changez un interrupteur local ou un interrupteur global, les pages de l'événement sont rafraîchies pour exécuter la bonne page.

La condition de déclenchement permet d'indiquer de quelle manière l'événement sera déclenché. Pour cela, vous utilisez la propriété `trigger` dans l'objet `page` :

### Configuration de l'événement – Déclenchement

```
{  
  "trigger": "action_button"  
}
```

Ici, l'événement sera déclenché si le joueur appuie sur la touche d'action (`Espace` par défaut) à côté de l'événement. Cette fonctionnalité est utile pour parler aux personnages non joueurs. Il existe d'autres types comme `contact` ou `auto`. Lisez la documentation pour plus de détails.

Vous avez créé un événement, encore faut-il l'ajouter sur la carte. Rappelez-vous que dans la méthode `loadMap`, vous avez une propriété `events`, qui est un tableau d'événements :

#### Ajout de l'événement sur la carte

```
rpg.loadMap('Map9', {
    tileset: '001-Grassland01.png',
    events: ['EV001'],
}, mapLoad);
```

Vous ajoutez l'événement `EV001` sur la carte `Map9`. Notez que `EV001` est le nom du fichier JSON et non celui de l'événement.

## Commandes événements

Un événement possède une liste de commandes à exécuter au déclenchement et qui sont propres à la page en cours. Quand une commande est terminée, l'événement passe à la suivante. Cela permet d'effectuer des actions précises lors de l'exécution de l'événement.

#### Configuration de l'événement – Commandes

```
[
  {
    "x": 16,
    "y": 7,
    "id": "1",
    "name": "EV001"
  },
  [
    "commands": [
      ...
    ]
  ]
]
```

Dans la page, on a la propriété `commands`. C'est un tableau contenant des chaînes de caractères, qui sont tout simplement les commandes. Dans notre exemple, nous allons afficher un texte et activer un interrupteur :

#### Contenu du tableau des commandes d'événements

```
commands: [
  "SHOW_TEXT: {'text': 'Bonjour'}",
  "SWITCHES_ON: 2"
]
```

Ici, on affiche le texte dans une boîte de dialogue et on active l'interrupteur n°2 après la fermeture de la boîte de dialogue. Une commande à la syntaxe suivante : "NOM\_EN\_MAJUSCULE: paramètre".

Le paramètre peut avoir le format JSON (c'est le cas pour la commande `SHOW_TEXT`). Dans ce cas, remarquez que le nom et la valeur sont entourés de simples guillemets. Le nom doit être en majuscules.

## Ajouter dynamiquement des événements

Ajouter des événements dynamiquement permet d'en ajouter autant que l'on souhaite sur la carte sans créer plusieurs événements de même type dans le dossier. Prenons l'exemple d'une bombe que le joueur place sur la carte à chaque fois qu'il appuie sur une touche.

Pour cela, nous devons préparer un événement, c'est-à-dire le stocker sans pour autant l'afficher. Deux méthodes s'offrent à nous :

- `prepareEventAjax`;
- `prepareEvent`.

La seule différence entre les deux est l'appel des propriétés de l'événement. Dans le premier cas, il va chercher en Ajax dans le dossier `Data/Events` comme pour le chargement de la carte. Dans le deuxième cas, on donne les propriétés directement en paramètres.

Nous allons privilégier la première méthode.

### REMARQUE Mobile et mémoire

On peut se demander si ce choix est pertinent pour du mobile, où chaque octet compte (ou presque). Néanmoins, l'événement n'est appelé qu'une seule fois et mis en mémoire ensuite.

### Préparer un événement

Il vous suffit donc d'utiliser la méthode `prepareEventAjax` avec le nom du fichier en paramètre (sans l'extension).

#### Récupérer les paramètres d'un événement en Ajax

```
rpg.prepareEventAjax("bombe");
```

Remarquez qu'une fonction de rappel est possible en deuxième paramètre (voir la documentation).

### Ajouter sur la carte

Lors d'une certaine action du joueur, vous voulez ajouter sur la carte l'événement préparé.

### Ajouter dynamiquement l'événement sur la carte

```
rpg.addEventPrepared("bombe");
```

Vous pouvez utiliser cette méthode aussi souvent que vous le voulez pour ajouter plusieurs événements sur la carte.

### Changer les propriétés de l'événement préparé

Les positions des bombes sont fixées dans les propriétés de l'événement. Pourtant, dans le jeu, les bombes sont posées en diverses positions. Il faut donc préciser cette information avant d'ajouter une nouvelle bombe sur la carte. Nous allons utiliser la méthode `setEventPrepared` :

### Changer les propriétés de l'événement avant de l'ajouter sur la carte

```
rpg.setEventPrepared("bombe", {x: 1, y: 12});
rpg.addEventPrepared("bombe");
```

Remarquez qu'une fonction pourrait rendre plus propre le code ci-dessus.

## Ajouter une animation

Pour l'exemple, nous allons insérer une animation nommée `coin`, dont les données sont stockées dans le fichier `Heal5.png`, dans `Graphics/Animations`.

### Taille d'une frame de l'animation

```
rpg.setGraphicAnimation(192, 192);
```

Avant tout, il faut définir la taille d'un modèle (*pattern*), c'est-à-dire la partie de l'animation sur notre image. Ici, nous définissons la taille de  $192 \times 192$  px.

On doit désormais ajouter l'animation pour pouvoir la réutiliser à volonté dans le jeu.

### Ajouter une animation

```
rpg.addAnimation({
  name: 'coin',
  graphic: 'Heal5.png',
  framesDefault: {y: 40, x: 40},
  frames: [
  ]
});
```

Il nous faut définir plusieurs propriétés :

- `name` : le nom unique de l'animation ;
- `graphic` : le nom du fichier image présent dans `Graphics/Animations` ;

- `framesDefault` : indique les propriétés toujours valables pour chaque frame de l'animation (ici, les `frames` seront toujours décalées de (40,40) px par rapport à leur point d'origine);
- `frames` : propriété plus intéressante puisqu'elle va définir les frames de l'animation.

#### ALLER PLUS LOIN

La documentation montre encore d'autres propriétés comme l'ajout d'un son.

Nous définissons dans un tableau les propriétés de chaque frame, par exemple :

#### Construction de l'animation

```
frames: [
  [{pattern: 13, zoom: 50}],
  [{pattern: 13, zoom: 80}, {pattern: 1, zoom: 20}],
  [{pattern: 13, zoom: 100}, {pattern: 2, zoom: 120}],
  [{pattern: 13, zoom: 120, opacity: 200}, {pattern: 3, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 170}, {pattern: 4, zoom: 120}],
  [{pattern: 13, zoom: 130, opacity: 150}, {pattern: 5, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 170}, {pattern: 6, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 200}, {pattern: 7, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 200}, {pattern: 7, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 170}, {pattern: 8, zoom: 120}],
  [{pattern: 13, zoom: 130, opacity: 150}, {pattern: 9, zoom: 120}],
  [{pattern: 13, zoom: 125, opacity: 170}, {pattern: 10, zoom: 120}],
  [{pattern: 13, zoom: 110, opacity: 200}, {pattern: 11, zoom: 120}],
  [{pattern: 13, zoom: 80, opacity: 180}, {pattern: 12, zoom: 120}],
  [{pattern: 13, zoom: 60, opacity: 180}],
  [{pattern: 13, zoom: 50, opacity: 150}],
  [{pattern: 13, zoom: 50, opacity: 100}],
  [{pattern: 13, zoom: 50, opacity: 50}]
]
```

Chaque élément est un tableau de plusieurs parties de l'image (pattern) :

- `pattern` : l'identifiant de l'animation;
- `zoom` : le zoom en pourcentage sur le pattern (50 diminuera l'image de moitié);
- `opacity` : l'opacité du pattern entre 0 (transparent) et 255 (opaque);
- `x` : décalage X par rapport à son origine. Remarquez que la propriété a déjà été définie dans `framesDefault`. Cependant, cela n'empêche pas la remettre ici. Si c'est le cas, elle sera prioritaire par rapport à la propriété dans `framesDefault`.
- `y` : décalage Y par rapport à son origine;
- `rotation` : tourne le pattern entre 0 et 360°.

L'enchaînement des frames va effectuer une animation.

## Afficher l'animation

Deux méthodes s'offrent à vous :

- par un événement;
- simple code.

### Un événement

Il vous suffit d'utiliser la commande d'événement : `show_animation`.

#### Commande d'événement pour afficher une animation sur le joueur

```
| {"SHOW_ANIMATION": {"name': 'coin', target: 'Player'}}
```

Lors du déclenchement de l'événement, l'animation nommée `coin` s'affichera sur le joueur. Remarquez que la propriété `target` peut aussi contenir le nom d'un autre événement.

### Simple code

Le code pour jouer une animation est très simple :

#### Jouer une animation

```
| rpg.animations['coin'].play();
```

Il vous suffit d'appliquer la méthode `play` sur l'animation en question. Remarquez que cette animation ne s'affiche peut-être pas à l'endroit que vous désirez. Pour définir les positions de l'animation, vous avez deux méthodes :

#### Positionner une animation sur la carte

```
| rpg.animations['coin'].setPosition(6, 5);  
| rpg.animations['coin'].play();
```

Ici, on affiche l'animation sur le carreau (6,5) de la carte.

#### Positionner l'animation sur l'événement

```
| rpg.animations['coin'].setPositionEvent(event);  
| rpg.animations['coin'].play();
```

De cette manière, on affiche l'animation sur un événement. Elle restera donc fixe sur l'événement, même si ce dernier bouge. Remarquez que le paramètre `event` est un objet `Event` et non le nom de l'événement. La documentation vous montre quelques méthodes pour récupérer un événement, par exemple `getEventByName()` pour l'obtenir par son nom.

## Créer des actions

Le joueur peut effectuer une action : attaquer, poser un objet, se défendre, tirer un objet, etc. RPG JS dispose d'une méthode pour ajouter des actions. Remarquez que l'action intervient le plus souvent à l'appui d'une touche, mais il est possible d'en déclencher aussi sur un événement (pour le combat A-RPG par exemple).

### Ajouter une action

Nous allons utiliser la méthode `addAction()` qui possède deux paramètres : le nom unique de l'action et ses propriétés. Nous allons prendre l'exemple d'un coup d'épée donné par le joueur quand la touche A est pressée.

#### Ajouter une animation

```
rpg.addAction('myattack', {  
    action: 'attack',  
    suffix_motion: ['_SWD_1'],  
    duration_motion: 1,  
    block_movement: true,  
    wait_finish: 5,  
    speed: 25,  
    keypress: [Input.A]  
});
```

Le nom de l'action s'appelle `myattack`. On aura besoin de ce nom plus tard pour l'affecter au joueur. Cette action présente les propriétés suivantes :

- `action` : utile seulement si cette action influence le combat. Poser un objet, par exemple, n'a rien à voir avec le combat et il serait inutile de mettre cette action. Ici, le coup d'épée attaque les ennemis : il est convenable d'attribuer `attack` à cette propriété.
- `suffix_motion` : la propriété la plus importante. C'est l'image dans `Data/Characters` qui s'affichera lors de l'action. Par exemple, si c'est le héros qui effectue cette action et que l'image par défaut se nomme `Hero.png`, cette propriété va chercher l'image `Hero_SWD_1.png` comme action. Remarquez que la propriété est un tableau. Il est possible d'en mettre plusieurs et de piocher un mouvement parmi eux (cette fonctionnalité est en cours d'amélioration - voir la documentation pour plus de détails).
- `duration_motion` : la durée du mouvement en frames. C'est le nombre de fois où l'animation du mouvement est effectuée. Par exemple, si vous mettez 3, on verra le héros donner 3 coups d'épée.
- `block_movement` : bloque le mouvement du joueur quand l'action est effectuée. En clair, vous ne pouvez plus bouger.
- `wait_finish` : le nombre de frames à attendre avant de réutiliser l'action. Dans le cas du coup d'épée, cela évite surtout de voir le joueur effectuer l'action trop vite.
- `speed` : rapidité de l'action. Plus la valeur est grande, plus l'action est rapide.
- `keypress` : les touches à presser pour déclencher l'action.

Cette méthode propose d'autres propriétés comme `callback` pour appeler une fonction et faire « quelque chose » quand l'action est déclenchée. Vous êtes invité à lire la documentation pour les autres propriétés.

### Affecter l'action

Ajouter une action au joueur est très simple. Rappelez-vous : quand vous chargez la carte, vous avez une propriété `player` avec ses positions X et Y, sa direction, le nom de l'image, etc. Hé bien, vous pouvez lui ajouter la propriété `actions` :

#### Affecter l'action au joueur

```
rpg.loadMap('MAP007', {
    tileset: '001-Grassland01.png',
    player: {
        x: 11,
        y: 2,
        filename: '001-Fighter01.png',
        actions: ['myattack']
    }
});
```

# C

## 3D avec Three.js

---

Three.js est une bibliothèque pour programmer de la 3D avec Javascript. Pour cela, il peut utiliser HTML5 [Canvas](#) pour des rendus peu complexes ou WebGL pour des animations reposant sur plus de performances.

Cette annexe ne prétend pas vous donner toutes les explications sur Three.js, mais offre une première approche sur cette bibliothèque avant de passer à l'étape supérieure : la 3D.

## Installation

Dans un premier temps, il faut installer Three.js. Téléchargez la dernière version sur Github, puis insérez la bibliothèque en en-tête de votre page.

▶ <https://github.com/mrdoob/three.js>

### Insertion de Three.js sur la page

```
<script src="Three.js"></script>
```

## Créer une scène

Une scène est composée de tous les éléments 3D. Pour la créer, nous avons besoin de trois objets lors de l'initialisation :

- `scene` : notre scène ;
- `camera` : la caméra avec une position ou une rotation ;
- `renderer` : le rendu à afficher sur l'écran selon la position de la caméra.

Nous initialisons les variables :

### Initialisation des variables `scene`, `camera` et `renderer`

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(80, 4/3, 0.1, 1000);
var renderer = new THREE.WebGLRenderer();
camera.position.z = 100;
```

La caméra virtuelle est composée de quatre paramètres :

- Le premier paramètre est le champ de vision, un angle exprimé en degré.
- Le second est le ratio entre la largeur et la hauteur du rendu (4/3, 16/9, etc.).
- Les troisième et quatrième paramètres définissent les éléments entre deux distances.

Tous les éléments sont positionnés au point d'origine (0,0,0). Ainsi, nous reculons la caméra sur l'axe Z pour pouvoir voir l'élément.

Ajoutons ensuite la caméra sur la scène :

### Ajouter la caméra

```
scene.add(camera);
```

La méthode `WebGLRenderer` crée un élément DOM que nous ajoutons sur la page. Nous définissons une taille qui est liée au ratio de la caméra.

### Taille du rendu et ajout sur la page

```
renderer.setSize(400, 300);
document.body.appendChild(renderer.domElement);
```

#### ASTUCE Utiliser seulement le Canvas 2D

Dans le cas où le navigateur ne reconnaît pas WebGL, vous pouvez utiliser le classique HTML5 Canvas :

```
var renderer = THREE.CanvasRenderer();
```

Cependant, la qualité et la performance seront diminuées.

## Ajouter un objet

Pour ajouter un objet, nous distinguons deux points :

- la matière ;
- la géométrie.

La matière de l'objet comprend la texture, la couleur, la réflexion, etc. La géométrie définit la forme : un cube, une sphère, etc.

#### Ajouter une sphère texturée

```
var material = new THREE.MeshLambertMaterial({
    map: THREE.ImageUtils.loadTexture("images/wood.jpg")
});
var sphere = new THREE.Mesh(new THREE.SphereGeometry(30, 10, 10), material);
scene.add(sphere);
```

Nous définissons une texture de bois dans les paramètres de la méthode `MeshLambertMaterial` pour la matière de l'objet.

La méthode `Mesh` possède deux paramètres :

- la forme : ici, c'est une sphère que nous souhaitons afficher ;
- la matière initialisée plus haut.

**Figure 20-1**  
Sphère



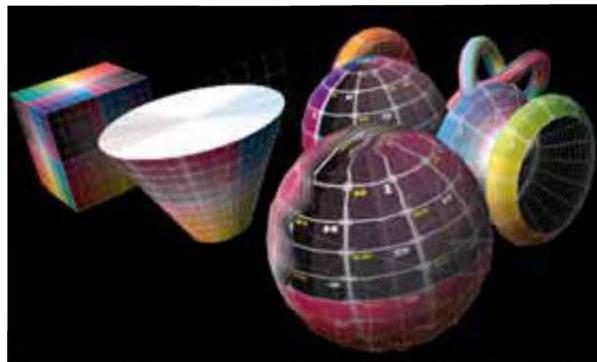
Pour la géométrie, vous avez aussi :

- `CubeGeometry : cube;`
- `CylinderGeometry : cylindre;`
- `IcosahedronGeometry : icosaèdre;`
- `PlaneGeometry : plan;`
- `TorusGeometry : anneau;`
- `TorusKnotGeometry : nœud.`

Nous vous invitons à lire la documentation de Three.js pour connaître les différents paramètres.

**Figure 20-2**

Quelques formes géométriques  
(Exemple tiré de Three.js)



## Former un groupe d'objets

Lorsque vous faites des manipulations sur le groupe, tous les éléments composant ce groupe seront modifiés.

### Créer un groupe

```
var group = new THREE.Object3D();
scene.add(group);
```

Utilisez la méthode `Object3D` pour créer un groupe et ajoutez-le à la scène. Pour ajouter un élément dans le groupe :

### Ajouter la sphère dans le groupe

```
group.add(sphere);
```

## Source de lumière

La source de lumière donne une ambiance à votre scène et affiche des ombres sur les éléments.

### Source de lumière

```
var light = new THREE.PointLight(0xFFFFFF);
light.position.set(20, 50, 100);
scene.add(light);
```

Nous utilisons la `THREE.PointLight` pour créer un point de lumière. Nous lui affectons la couleur blanche (en hexadécimal) et nous le positionnons à (X=20, Y=50, Z=100).

L'ajout sur la scène est équivalent à celui des objets.

#### ASTUCE Affecter des positions

C'est la méthode `set` qui affecte des positions, mais nous pouvons aussi passer par des propriétés :

```
light.position.x = 20;
light.position.y = 50;
light.position.z = 100;
```

**Figure 20-3**

Notre sphère avec une ombre causée par la source de lumière



## Rendu

Pour jouer en boucle les animations sur la scène, nous allons utiliser la méthode `requestAnimationFrame`.

### Rotation continue de la sphère

```
animate();
function animate() {
    sphere.rotation.y += 0.01;
    renderer.render(scene, camera);
    requestAnimationFrame(animate);
}
```

La fonction `animate` est amorcée dès le départ et ensuite appelée en boucle pour effectuer une rotation perpétuelle de la sphère sur elle-même.

Du coup, comme vous pouvez le constater, tous les objets déclarés précédemment – la caméra, les formes géométriques, un groupe d'objet, la source de lumière – peuvent être modifiés en changeant leurs propriétés.

#### ATTENTION Si vous chargez une texture

Si votre objet possède une texture, pensez à charger l'image avant de lancer l'animation :

```
var img = new Image();
img.onload = function() {
    animate();
};
img.src = "images/wood.jpg";
```

## Bouger la caméra avec la souris

Pour contrôler la caméra avec la souris, Three.js nous offre une classe `TrackballControls`.

#### Création d'un contrôleur

```
var controls = new THREE.TrackballControls(camera);
controls.target.set(0, 0, 0);
```

Nous souhaitons bouger la caméra lorsque nous maintenons le clic gauche. Nous plaçons la caméra au point d'origine (0,0,0). Ainsi, si notre sphère se trouve au même point, nous tournerons autour d'elle.

Évidemment, il ne faut pas oublier de mettre à jour le contrôleur dans le rendu :

#### Mise à jour du contrôleur dans le rendu

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

# Index

---

## Symboles

3D  
Three.js 232

## A

adversaire  
affichage des dommages 124  
barre de vie 122  
calcul des dégâts 121  
champ de vision 127  
paramètres 118  
réaction 128  
suppression 123  
zone de détection 127  
zone d'interaction 124  
ambiance du jeu 111  
animation 48, 51, 139  
déformation 48  
Ease (classe) 49  
en boucle 49  
réaction à une action 52  
temporaire 53

## B

barre de progression 18, 19  
bouton  
affichage 23  
événement 24  
initialisation 22

## C

canvas HTML5  
afficher un texte 204  
arc 200  
charger 198  
chemin 201  
composition 206  
couleur 205  
couper une image 208  
dégradé linéaire 203  
dégradé radial 203  
dessiner 199  
initialiser 198  
insérer une image 207  
ligne 199  
manipulation de pixels 212  
ombre 205  
recadrer 202  
rectangle 201  
redimensionnement 211  
redimensionner une image 208  
répéter une image 209  
rotation 210  
transformation personnalisée 211  
translation 210  
carte du jeu 29  
affichage 32, 39  
création 38  
Tileset 30, 32, 39  
collision 141  
concept du jeu 1  
condition binaire 92

contrôle 61  
 accélération 71, 72  
 accéléromètre 65  
 bouton de manette 69  
 clavier 63, 64  
 décélération 71, 73  
 écouteur clavier 73, 79  
 gravité 75, 77  
 joystick de manette 70  
 manette 67  
 saut 75, 77  
 souris 64  
 création d'un niveau 138  
 créer un modèle de données 28  
 créer un niveau 34  
 schéma de données 35

**D**

décor du jeu 29  
 affichage 32, 39  
 création 38  
 Tileset 30, 32, 39  
 défilement 140  
 classique 82  
 différentiel 83  
 du décor 81  
 deviceorientation (écouteur) 67

**E**

Easel.js  
 canvas en boucle 217  
 conteneur 218  
 forme 216  
 image 219  
 installation 216  
 texte 217  
 écran  
 niveaux 26  
 options 25  
 titre 15, 21  
 effet graphique 111, 115  
 effet jour/nuit 115  
 flash visuel 115

effet sonore 111, 112  
 fondu musical 114  
 HTML5 Audio 112  
 SoundManager 112, 113  
 état d'un élément 58  
 étude de marché 2  
 événement 24, 157

**F**

Facebook 177  
 SDK 178  
 framework 11  
 Caat.js 13  
 CanvasEngine.js 13, 14  
 Crafty.js 13  
 Easel.js 11, 216  
 Kinetic.js 12  
 RPG JS 220

**G**

Gamepad.js 68, 69  
 Gameplay 57  
 gravité 143

**I**

image  
 chargement initial 16  
 charger 20  
 qualité 16  
 Image (classe) 20  
 initialisation 136  
 Input (classe) 61  
 interaction personnage/décor  
 bord de la carte 87  
 collision 86, 87, 89, 90, 93  
 déclenchement automatique 96  
 déclenchement sur action 99  
 Hitbox 86  
 sur des objets 89, 93  
 sur le décor 89, 90

**J**

jeu  
 authentification dans Facebook 173  
 autorisation dans Facebook 173  
 composer la structure 158  
 configuration du paiement 193  
 déclaration dans Facebook 172  
 jouabilité 57  
 joueur ciblé 6  
 JSON 17, 32, 133, 221

**K**

keyDown 63  
 keyPress 63

**L**

level design 28  
 ligne temporelle virtuelle 48

**M**

Math.max 122  
 Math.min 122  
 monétisation 191  
 MongoDB 169  
 monnaie virtuelle 192  
 mouvement 58, 143, 145  
 multijoueur  
   afficher un badge 185  
   afficher un score 183  
   configurer le serveur 150  
   connexion 180  
   connexion/déconnexion 164  
   gestion des données 162, 163  
   intégration du SDK Facebook 178  
   inviter des amis 181  
   Node.js 150, 151, 155, 159  
   NPM (Node Packaged Modules) 152  
   récupérer des informations 186  
   Social Gaming 177  
   Socket.io 152, 156, 164  
   statut du joueur 179  
   tester l'installation 153

**N**

Node.js 155  
 module 159

**P**

pause 108  
 plate-forme mobile 58  
 position de la souris 25  
 positionnement du jeu 6  
 prix du jeu 8

**Q**

qualité d'image 16

**R**

rafraîchissement 19, 22  
 règle du jeu 101, 136  
   affichage des points 104  
   affichage du score 108  
   application du concept 104  
   boîte de dialogue 106  
   fin de partie 109  
   inventaire 103  
   situation initiale du joueur 102  
 RPG JS  
   action 229  
   animation 226  
   carte 221, 222  
   événement 222, 224, 225, 226  
   jeu de rôle 220

**S**

saut 143  
 sauvegarde 133  
   chargement des données 133  
   Marshal (classe) 132, 134  
   sérialisation des données 132  
 Scrolling 81  
 Scrolling (classe) 82  
 situation initiale du joueur 102  
 Social Gaming 177  
 Socket.io 156