



EN BONUS

- ★ Des outils pour accélérer vos développements
- + la réalisation pas-à-pas de 2 jeux vidéos !

L12225 - 25 H - F. 8,00 € - RD



DÉCOUVREZ LE WEB DU FUTUR !

HTML

... OU COMMENT EXPLORER LES SPÉCIFICITÉS DU LANGAGE TOUT EN CRÉANT VOTRE PROPRE JEU POUR LE WEB ?



1 ON PLANTE LE DÉCOR

- ★ Quelques rappels sur le game design
- ★ Les solutions pour monétiser son jeu
- ★ Modéliser son plateau de jeu
- ★ L'importance des graphismes

2 SE FAMILIARISER AVEC HTML 5

- ★ Bien utiliser les canvas et animer les sprites
- ★ Intégrer des éléments vectoriels (SVG)
- ★ Gérer les contrôles clavier/souris
- ★ Utiliser des polices externes

3 PEAUFINER SON JEU VIDÉO

- ★ Gérer les données persistantes
- ★ Proposer un mode multijoueur
- ★ Ajouter une bande son
- ★ Créer ses décors avec un éditeur de niveaux

Actuellement
en kiosque !

GNU/Linux Magazine N°153

Systemd PRÊT À REEMPLACER INIT ... OU PAS

DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX
JUSQU'AU **26 OCTOBRE** 2012
ET SUR :
www.ed-diamond.com



N°153 OCTOBRE 2012

LINUX MAGAZINE / FRANCE

ADMINISTRATION ET DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE ET EMBARQUÉS

NETADMIN / WINDOWS

Remplacez enfin vos serveurs Windows grâce à SAMBA 4 et son support complet de l'Active Directory !

p.43

RADIO / LABO

Explorez la Software Defined Radio (SDR) à peu de frais avec GNU Radio et un récepteur TNT p.04

CODE / C

Retour sur le problème de la seconde intercalaire et son impact sur votre système p.80

ANDROID / SYSTÈME

Découvrez comment Android gère le système graphique du matériel à SurfaceFlinger p.64

DÉMARRAGE / SERVICES

Systemd
PRÊT À
REEMPLACER INIT
... OU PAS p.34

WEB / BENCH

Évaluez concrètement les performances effectives de vos applications web avec Jmeter p.58

SAUVEGARDE / DATA

Sauvegardez et clonez à chaud vos machines en toute simplicité avec MondoRescue p.50

CODE / PHP

Elaborez un système de cache sur les méthodes des objets avec les méthodes magiques de PHP p.75

L 19275 - 15 - F 7,50 €

France METRO : 7,50 € - CH : 13 CHF - BELGIQUE CONT : 1,80 € - DOM : 8 € - CAN : 13,75 \$ cad - NOUVEL : 10,00 CFP - POLOGUE : 11,00 CFP - POLUA : 11,00 CFP - TUNISIE : 14,50 THD - MAR : 14 MAD

DÉCOUVREZ UN NOUVEAU LANGAGE DANS
GNU/LINUX MAGAZINE HORS-SÉRIE 63 !

le 2 novembre
en kiosque !

GO, LE LANGAGE DE PROGRAMMATION
DÉVELOPPÉ PAR GOOGLE

- Toutes les bases pour commencer (historique, compilation, syntaxe de base, ...)
- Des explications sur les différents types de variables (chaînes, pointeurs, tableaux, ...)
- Et pour aller + loin : les fonctions, la programmation orientée objet, le traitement des fichiers, les tests unitaires, ...

Un numéro complet pour explorer
et maîtriser le langage Go !

le 2 novembre 2012 chez votre marchand de journaux

édito

Ah, enfin ! Fini la plage, le soleil et le farniente ! Nous allons enfin pouvoir nous enfermer à nouveau pour développer ! Mais peut-être que les vacances ne se sont pas encore totalement dissipées ? Peut-être vous reste-t-il encore quelques souvenirs nostalgiques de jeux de ballon sur la plage... Tiens, des jeux... Et si justement vous dévelopez un jeu ? Et si, soyons fous, vous alliez plus loin et que vous dévelopez un jeu Web ? Je ne vous parle pas de ces vulgaires jeux en Flash, mais de véritables jeux ! Un jeu d'action, un jeu de stratégie ou d'aventure ?

Dans ce hors-série, nous ferons un point sur les techniques de programmation HTML 5 qui vous permettront de parvenir à vos fins. Avec l'avènement des smartphones, nous assistons à la naissance de dizaines de jeux plus intéressants les uns que les autres... et malheureusement, la tendance est la même sur les autres plateformes. Nous nous pencherons sur ce phénomène en tentant de comprendre qu'est-ce qui fait un bon jeu, un jeu qui va marquer toute une génération et auquel on pourra encore faire référence des années après avec nostalgie. Ceux d'entre vous qui ont eu la chance de jouer à des jeux comme « Maniac Mansion », « Secret of Monkey Island », « Speed Ball », « Zelda », « Diablo », « Dungeon Keeper », « Baldur's Gate », etc. comprendront de quoi je parle.

Alors s'agit-il d'un réflexe de repli sur soi ? Le regret de notre vie « d'avant », lorsque nous étions jeunes et que nous pouvions passer des heures et des heures scotché devant l'écran à jouer (oui, parce que pour la majorité d'entre nous, je pense que nous sommes quand même scotché devant notre écran, mais pour d'autres raisons...). En d'autres termes, pourrions-nous parler du réflexe dit « du vieux con » ? Aigris par le rythme infernal de nos vies, nous allons critiquer des jeux pour lesquels nous n'arrivons pas à trouver le même attachement que par le passé : « trop facile », « trop court », « un mode solo bâclé au profit du multi-joueur » et la plupart du temps « trop cher » ! Car il ne faut pas l'oublier, nous parlons ici de jeux et seulement de jeux ! Même si les moyens mis en œuvre pour les développer sont importants, ils restent dans le domaine du divertissement et de plus en plus souvent, un divertissement à une durée de vie relativement faible ! Les éditeurs de jeux semblent oublier ce point. Heureusement qu'il reste les éditeurs de jeux indépendants ! Si vous ne connaissez pas encore les *humble bundle*, vous pouvez vous rendre sur le site <http://www.humbledumble.com> pour découvrir des jeux vendus sous la forme de paquetages au profit d'associations caritatives. Ces jeux sont bien souvent très intéressants (dans un humble dumble précédent, on a ainsi pu voir le désormais célèbre « World of Goo ») et, point non négligeable, ils fonctionnent sous votre OS préféré !

Tous les auteurs de ce hors-série ont pris beaucoup de plaisir à écrire les articles... tout en restant quelque peu frustrés par l'impossibilité d'aller plus loin et de réaliser leurs propres jeux, le temps leur manquant cruellement ! J'espère que le temps ne vous fait pas défaut, que les différents conseils et techniques exposés dans ce magazine vous seront profitables et que nous pourrons bientôt tester vos jeux !

Tristan Colombo



www.linux-pratique.com

Linux Pratique Hors-Série
est édité par **Les Éditions Diamond**

B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@linux-pratique.com

Service commercial :
abo@linux-pratique.com

Sites : www.linux-pratique.com
www.ed-diamond.com



Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédactrice en chef : Fleur Brosseau
Secrétaire de rédaction : Véronique Sittler
Conception graphique : Kathrin Scali
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Valérie Fréchard, Tél. : 03 67 10 00 27, v.frechard@ed-diamond.com
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Illustrations : www.fotolia.com
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou, Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier, Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 2101-6836

Commission Paritaire : K78 990

Périodicité : Bimestrielle

Prix de vente : 8,00 Euros

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Linux Pratique Hors-Série est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à Linux Pratique Hors-Série, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire. Toutes les marques citées dans ce numéro sont déposées par leur propriétaire respectif. Tous les logos représentés dans le magazine sont la propriété de leur ayant droit respectif.

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

sommaire

LA THÉORIE

- 04 Le « game design », ou comment construire son jeu ?
- 08 Monétisation des jeux libres (parce que vivre de code et d'eau fraîche, ce n'est pas plaisant qu'un moment)
- 12 Et si on jouait cartes sur table ?
- 17 Détection des collisions
- 20 L'importance du graphisme
- 22 HTML 5/CSS 3 : pourquoi peut-on maintenant créer des jeux web ?

LA TECHNIQUE

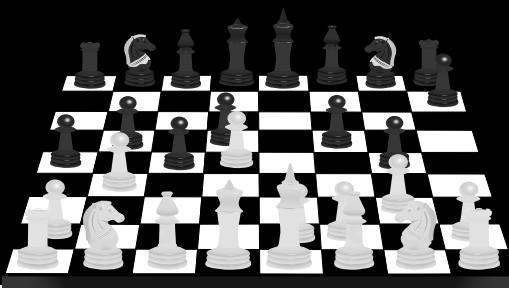
- 24 Graphismes : utilisation des canvas
- 30 Animer un sprite
- 35 HTML 5 et SVG
- 39 Les contrôles au clavier et à la souris
- 44 Utiliser des polices de caractères externes
- 48 Gérer des données persistantes
- 50 Le mode multijoueur
- 57 Un peu de musique dans ce monde de brutes
- 62 Éditeur de niveaux

LA PRATIQUE

- 64 Des outils pour accélérer le développement
- 70 Un jeu de morpion en HTML 5
- 80 Créer un jeu complet, avec carte isométrique

15 / 55 / 56 ABONNEMENTS & COMMANDES





LE « GAME DESIGN », ou comment construire son jeu ?

par Jean-Michel Armand

Les jeux vidéo, il y a les bons et les mauvais. Dire *a posteriori*, une fois que le jeu est fini, en boîte, qu'il est bon ou mauvais, même si cela reste subjectif, est assez facile, même en prenant en compte le fait d'argumenter sa position. Ce qui est plus difficile, c'est de ne pas concevoir son jeu en aveugle. Ce qui est compliqué c'est d'intellectualiser sa démarche créative pour améliorer son jeu. C'est à cela que sert le game design. Cet article tentera de vous donner quelques clés qui vous aideront, je l'espère, lors de la création de votre jeu.



1. Le jeu, vos joueurs et vous

Avant d'étudier en détails les mécanismes d'un jeu, il me semble intéressant de revenir sur quelques notions de base. Vous vous apprêtez donc à créer un jeu vidéo. Ce jeu sera le seul lien avec votre public. Vous ne savez en effet *a priori* rien sur ceux et celles qui vont y jouer. Est-ce que ce seront des femmes ou des hommes ? Des trentenaires ou des étudiants ? Des joueurs occasionnels ou au contraire des joueurs assidus ? Vous ne le saurez pas, en tout cas pas dans un premier temps. Et pourtant, il vous faudra concevoir un jeu qui plaira à vos joueurs, sur lesquels vous ne connaissez rien. Et plus que simplement leur plaisir, il faudra à travers votre jeu leur faire vivre une expérience ludique qu'ils aimeront.

Mais comment créer un jeu qui générera une expérience plaisante pour des personnes que vous ne connaissez pas ? Je vais vous donner quelques pistes dans très peu de temps, mais avant cela, je voulais préciser une chose. Ce n'est pas le jeu en lui-même qui est l'expérience. Le jeu, à l'aide de ses règles et ses composantes n'est que le moyen de faire vivre à vos joueurs l'expérience

ludique. Le jeu n'est pas du tout une fin en soi, ce n'est qu'un média. Lorsque vous pensez à votre jeu, il faut donc le penser dans cette optique-là. Quelle est l'expérience que vous voulez délivrer à vos joueurs ? Qu'est-ce qui est important, essentiel à cette expérience ? Quels sont les éléments de votre jeu qui aident à cela ? Et c'est comme cela, en pensant à l'expérience de vos joueurs et pas à votre jeu, que vous pourrez dans un premier temps énoncer clairement l'expérience que vous voulez offrir à vos joueurs et que, dans un deuxième temps, vous pourrez instiller celle-ci dans votre jeu.

Mais comment définir cette expérience dont je parle depuis un paragraphe ? Il faut déjà, à mon avis, définir à traits grossiers qu'elle est votre cible de joueurs. Et - mais ce n'est que mon avis - je ne vous conseille pas de choisir comme cible « les gens comme moi ». Parce que même si je n'ai rien contre les jeux de niches, bien au contraire d'ailleurs, j'en suis un grand fan, la niche « les gens qui sont mes quasi-clones » me semble être un peu trop petite pour que votre jeu représente un investissement valable. Mais encore une fois, ce point précis n'est que mon avis.

Définir à traits grossiers sa cible n'est bien entendu pas suffisant. Il va falloir

comprendre comment les expériences merveilleuses et inoubliables naissent. Pour cela, il faudrait rien de moins que comprendre comment se créent émotions et sentiments. Tâche difficile s'il en est et qui nous éloignerait un peu de la création de jeu vidéo. Il faudra se contenter de moyens un peu plus à notre portée, l'introspection. Il ne faudra toutefois pas que vous vous contentiez d'analyser vos réactions vis-à-vis de votre jeu en devenir, il faut aussi que vous tentiez de vous mettre à la place de vos joueurs, pour ensuite tenter d'imaginer et d'analyser leurs probables réactions.

Avant de poursuivre plus avant dans le décorticage des composantes d'un jeu vidéo, une dernière chose. Nous venons de voir que les joueurs doivent vivre une expérience par l'intermédiaire de votre jeu. Et c'est la qualité de cette expérience qui définira s'ils passeront un bon moment en compagnie de votre jeu et donc, s'ils y joueront. Quelles sont les émotions qui vont faire que vos joueurs vont passer un bon moment ? La première d'entre elles semble triviale. C'est la sensation d'amusement. Cela semble tellement évident que vous pourriez avoir l'impression que j'enfonce des portes ouvertes. Mais combien d'entre vous se sont-ils ennuyés dans un jeu ?



N°74

■ Retour sur la structure d'une page web en HTML 5

■ Étudiez les statistiques de votre site avec AWStats

■ Les outils pour gérer vos bases de données



Et aussi :

- Le test du Raspberry Pi
- i3, un tiling window manager simple à utiliser
- La gestion de machines virtuelles à distance
- Des solutions pour chiffrer vos données sensibles
- Le traitement de chaînes en C sans douleur avec Bstring
- Le dessin assisté par ordinateur sous Linux
- Déploiement du système de fichiers Btrfs

Sous réserve de toute modification.

**CHEZ VOTRE MARCHAND DE JOURNAUX
DÈS LE 26 OCTOBRE**

Voire même, ennuyés jusqu'à ne plus avoir envie de jouer ?

Après l'amusement, il y a la surprise. La surprise relancera l'intérêt de vos joueurs, leur donnera envie d'aller plus loin pour essayer de trouver d'autres surprises. La curiosité est un autre moyen à utiliser pour faire naître l'intérêt chez votre joueur. Votre jeu devrait faire en sorte que les joueurs se posent des questions et surtout, qu'ils aient envie d'y répondre. Imaginons par exemple que vous développiez un jeu de casse-briques. Vous pourriez utiliser une image différente pour les fonds de chaque niveau. Vos joueurs ne joueraient pas uniquement pour finir les différents niveaux, mais aussi pour découvrir les images cachées par les briques.

La sensation de possession est aussi une sensation agréable pour vos joueurs ; après tout, les joueurs sont tous d'horribles capitalistes ; il vous faut donc définir quels sont les éléments dans votre jeu qui ont une valeur pour le joueur et définir comment ces éléments seront gagnés et perdus. Le mieux étant de réussir à mettre en relation la valeur dans le jeu et les motivations de vos joueurs. Enfin, la dernière émotion que vous pouvez utiliser pour améliorer l'expérience de vos joueurs est celle qui naît du plaisir de résoudre des énigmes ou des problèmes. Il suffit de voir le succès des jeux de Sudoku pour voir que résoudre des énigmes est quelque chose qui plaît.

2. Anatomie d'un jeu vidéo

Nous avons vu précédemment que le seul lien entre vous concepteur et vos joueurs serait le jeu que vous êtes en train de concevoir. Mais de quoi exactement sera constitué votre jeu ?

2.1 Les quatre éléments de base

Un découpage assez souvent utilisé pour les éléments d'un jeu est le découpage suivant :

- Mécaniques
- Esthétique

- Histoire

- Technologie

Les mécaniques sont les règles de votre jeu. Elles décrivent les buts de votre jeu, les moyens de tenter d'y arriver, les choses qui sont interdites et celles qui sont permises. Tout ce qui se passe dans un jeu est régi par l'une des règles définies dans les mécaniques de votre jeu.

L'esthétique correspond à tout ce qui va parler aux sens de vos joueurs. C'est-à-dire, c'est la dimension graphique, sonore, olfactive, etc., de votre jeu. L'esthétique est une dimension très importante de la création des jeux. On dira d'un jeu qui a une esthétique réussie qu'il a son style propre.

L'histoire correspond aux événements qui se dérouleront tout au long de votre jeu, ainsi qu'aux liens qui existent entre eux.

La technologie correspond à tous les matériels ou technologies que vos joueurs devront utiliser pour jouer à votre jeu. Cela peut être leur ordinateur, leur téléphone ou des périphériques comme la Wiimote ou Kinect. C'est également le fait d'avoir choisi de faire votre jeu en HTML 5. La technologie choisie pour votre jeu vous offrira des possibilités et vous en interdira d'autres.

2.2 Un jeu, c'est à la fois un univers et un thème

Ceci est vrai quel que soit le jeu que vous voulez mettre en place. Et pour être plus précis, c'est la mise en place de votre thème que vous allez déployer dans l'univers que vous avez choisi. Le thème c'est un peu comme la périphrase qui résumerait tout votre jeu, c'est la chose qui fédère chaque partie de votre jeu autour de lui. Décrire le thème de votre jeu, c'est un peu comme faire un *elevator pitch* pour un projet de création d'entreprise. Vous devez pouvoir expliquer votre thème en quelques secondes et quelques dizaines de mots.

Même si on peut très bien attendre qu'un thème émerge de lui-même du processus de création du jeu, il est assez intéressant de connaître le thème de son jeu rapidement. Cela vous permet en effet de savoir à tout instant

si l'élément que vous souhaitez ajouter a sa place dans votre jeu ou pas. Est-ce que cet élément renforce votre thème ? Si oui, alors il a sa place, si ce n'est pas le cas, c'est qu'il est superflu et qu'il ne faut pas l'ajouter. Une fois votre thème choisi, il se mettra en place dans un univers précis, celui que vous aurez choisi. Imaginons que nous choisissions comme thème « prendre la place d'un pilote de vaisseau spatial ». Il nous faudra choisir l'univers dans lequel on voudra déployer notre thème. Et, en fonction de l'univers choisi, notre jeu sera différent du tout au tout. Être un pilote de vaisseau spatial dans l'univers de Warhammer 40K n'a pas grand-chose à voir avec être un pilote de vaisseau spatial dans celui de Star Trek.

2.3 C'est aussi une histoire et des personnages

Un jeu peut ne pas avoir d'histoire ou de personnages. C'est le cas dénormément de jeux « casual ». Mais je trouve que c'est très souvent une erreur que de ne pas au moins essayer d'intégrer ces deux éléments dans un jeu. Parce que ce sont deux éléments qui renforcent à la fois votre thème et l'expérience de vos joueurs d'une façon exponentielle.

L'histoire va vous permettre d'offrir un contexte à votre jeu, de justifier les objectifs et les obstacles que votre joueur devra atteindre. Les personnages, quant à eux, vont ajouter de la vie à votre jeu, permettre d'instaurer un dialogue entre le joueur et votre jeu. Mais avoir une histoire et des personnages ne fait pas tout. Il faut que tout ce petit monde soit crédible.

Faire de bons personnages n'est pas aussi facile que l'on pourrait penser. Partir de leur fonction pour les définir est une bonne idée, mais ce n'est pas suffisant. Des personnages crédibles demanderont à ce que vous définissiez leur caractère. Il faudra les doter d'un passé et trouver pour quelle raison ces personnages déboulet dans l'histoire de votre jeu. Pourquoi cet astropilote rejoint votre équipage ? Quelles sont les raisons qui poussent le capitaine de l'astroport à vous détester ?

Vos joueurs se poseront la question et si vous ne leur donnez pas de réponse plausible, ils n'accrocheront pas. Mais n'oubliez pas que vos personnages sont plus que quelques lignes d'explications dans votre histoire. Un personnage c'est aussi un visuel et une voix. Malheureusement, la dimension sonore des personnages est souvent au mieux négligée quand elle n'est pas totalement oubliée. Donnez une voix à vos personnages, ils n'en seront que plus réels et vous augmenterez la qualité de l'expérience de vos joueurs.

3. Mécaniques et règles

Pour conclure cet article, je vais parler de mécaniques et de règles. Les règles constituent ce que l'on appelle souvent « le moteur » de votre jeu. Ce sont elles qui en définissant ce que le joueur peut ou ne peut pas faire, définiront le cadre de son expérience ludique. On pourrait voir les règles comme les fondations et les murs d'une construction. C'est en s'appuyant sur elles que vous construirez votre jeu. Pour une maison, si l'on veut que les choses soient solides, il faut que les fondations soient stables. Pour un jeu, si l'on veut que l'expérience de jeu soit agréable, il faut avoir des règles qui soient équilibrées. Et je ne parle pas ici d'un « simple » équilibrage des possibilités entre joueurs lorsqu'on se trouve dans un jeu multijoueur. Je parle d'équilibre entre les différents éléments des règles. Étudions quelques-uns de ces équilibrages.

- Challenge vs succès

Tous les jeux proposent des challenges à leur joueur. Et chaque fois qu'un joueur réussit un challenge, il remporte une récompense et se voit proposer un nouveau challenge, plus difficile. Si la difficulté des challenges n'augmente pas suffisamment, vos joueurs vont s'ennuyer ; si par contre ils deviennent trop rapidement très ardu, vous allez démotiver votre public.

- Pantouflard vs tête brûlée

Certains joueurs préféreront jouer d'une manière sécurisée, en prenant

peu de risques, même s'il savent que du coup ils gagneront peu. D'autres seront des risque-tout, qui tenteront le destin s'ils peuvent ainsi empêcher la mise maximale. N'oubliez pas de penser à ces deux possibilités et de proposer de tels choix à vos joueurs !

- Récompense vs punition

Tout le monde aime être félicité et récompensé. Posez-vous la question de savoir quelles sont les récompenses que votre jeu offre ? Sont-elles en adéquation avec le défi proposé au joueur ? Est-ce qu'il n'y en a pas trop, ou au contraire, pas assez ? Il peut paraître étrange de parler de punition dans un jeu vidéo, mais en fait, la punition est un mécanisme très utile pour augmenter l'expérience ludique, à condition qu'elle soit bien utilisée. En effet, qui dit punition dit risque, et prendre des risques est quelque chose d'excitant, qui donne une autre saveur au challenge. Toutefois, pour ne pas faire fuir vos joueurs lorsque vous intégrez un mécanisme de punition dans votre jeu, il faut que celui-ci soit clair et compréhensible. Il faut que votre joueur aient un moyen de savoir pourquoi, comment et puisse trouver une solution pour éviter les choses.

4. Conclusion

J'arrive au terme de cet article et je n'ai fait qu'effleurer et encore, de très très loin, ce que pouvait être le game design et quelques-unes des techniques que l'on pouvait utiliser pour construire un bon jeu. Heureusement pour vous, si le sujet vous intéresse, vous n'aurez que l'embarras du choix parmi toutes les ressources disponibles. Une dernière chose, avant de vous laisser tourner la page et commencer le prochain article : la création de jeu vidéo, c'est comme le développement, les choses s'améliorent par itération et *refactoring* successif et pour être sûr que tout fonctionne, il n'y a qu'une méthode, il faut tester ! Mais par contre le *Test Driven Game Design* n'existe pas encore, en tout cas à ma connaissance. Il vous faudra faire vos tests après, à l'ancienne ! ■

1&1 DOMAINES

**INCLUS : ADRESSE EMAIL
PERSONNALISÉE !**

FAITES-VOUS UN NOM !



FAITES CONFIANCE AU N° 1

Avec plus de 11 millions de contrats clients, 2 milliards de chiffre d'affaires en 2011, 5000 employés et 5 centres de données haute performance en Europe et aux Etats-Unis, nous comptons parmi les leaders mondiaux de l'hébergement. A cela s'ajoutent nos 18 millions de noms de domaine enregistrés, qui nous permettent de vous offrir des prix bas toute l'année.

✓ GESTION DNS

Contrôle total des paramètres DNS de vos noms de domaine.

✓ TOUT INCLUS

Adresse email personnalisée, 1000 alias, 10 sous-domaines.

✓ APPLI 1&1 NOMS DE DOMAINE

Recherchez la disponibilité des noms de domaine et réservez-les depuis votre mobile.

✓ SERVICE EXPERT 6J/7

Assistance assurée par des experts, via hotline non surtaxée et email.

.fr
.com
.eu
.net
.org
.info

à partir de
0,99
€ HT/an
(1,18 € TTC/an)*

Inclus avec tous les domaines 1&1 :

- Redirection de nom de domaine
- Gestion des données de contact
- Création et redirection de sous-domaine

1&1

DOMAINES | EMAIL | HÉBERGEMENT | E-COMMERCE | SERVEURS



0970 808 911 (appel non surtaxé)

www.1and1.fr

*Offre applicable la 1^{re} année uniquement sur une sélection de domaines. Ensuite, le prix habituel s'applique. Conditions détaillées sur 1and1.fr.



MONÉTISATION DES JEUX LIBRES

(parce que vivre de code et d'eau fraîche, ce n'est plaisir qu'un moment)

par Jean-Michel Armand

Sans compter qu'aujourd'hui, même l'eau fraîche a un coût. La monétisation d'un jeu (mais plus largement d'un logiciel) libre est donc une préoccupation que l'on a le droit d'avoir à l'esprit. Entendons-nous bien, il n'y a rien de mal à ne pas vouloir monétiser ses jeux libres et à aimer les faire vivre juste pour le plaisir. Mais de la même manière, il n'y a rien de mal à vouloir gagner un peu d'argent avec son jeu libre. Et nous allons voir comment on peut y arriver dans cet article.



Pour commencer, je vais définir clairement ce que j'entends par un jeu libre. Un jeu libre est pour moi un jeu dont le code source est libre, bien entendu, mais aussi dont suffisamment de ressources graphiques, sonores, visuelles sont libres pour que l'on puisse jouer uniquement avec celles-ci. Un jeu dont seul le code est libre, mais qui ne propose aucune ressource libre est un jeu auquel on ne peut jouer sans y adjointre une partie non libre. Ce n'est donc pas à mon sens un jeu libre. Ce petit point de définition éclairci, nous pouvons nous attaquer au vif du sujet.

Il y a énormément de moyens de monétiser un jeu libre. Certains plus faciles à mettre en place que d'autres, certains plus rémunérateurs que d'autres. Afin de les présenter d'une manière lisible,

Pour en savoir plus

La monétisation des projets libres et plus spécifiquement des jeux libres est un sujet qui me tient très fortement à cœur. Pour ceux intéressés par le sujet, j'ai donné une conférence aux Rencontres Mondiales du Logiciel Libre 2012 [1] sur le métier d'éditeur de logiciels libres et les moyens de gagner de l'argent en étant éditeur et cela, sans perdre ni transiger avec ses convictions.

nous les avons classés en fonction d'un critère simple : qui paie ? Cette classification permet de dégager trois grands groupes de modèles possibles :

- aucun joueur ne paie,
- certains joueurs paient,
- tous les joueurs paient.

Je finirai ensuite par un ensemble de moyens de financement qui ne rentrent pas vraiment dans les trois catégories listées au-dessus, mais que je trouve intéressant de détailler.

1. Aucun joueur ne paie, ou le miroir aux alouettes 2.0

On va retrouver dans ces types de monétisation possibles tous les modèles qui tournent autour de la maxime bien connue « si vous utilisez gratuitement un produit, alors c'est que vous êtes le produit ». Je veux bien entendu parler de la publicité. Bien que ce modèle soit a priori le plus aléatoire et le moins rémunérateur, c'est aussi celui auquel les créateurs pensent le plus souvent.

Personnellement, je n'aime pas ce modèle de rémunération. Pour deux raisons. La première c'est que bien souvent il est mis en place sans aucune

subtilité, et que jouer à un jeu où il y a à droite et à gauche de la zone de jeu deux colonnes de publicités clignotantes (et sans rapport aucun avec le jeu en lui-même) détruit toute l'immersion que peut offrir le jeu et nuit vraiment fortement au plaisir de jouer.

La seconde c'est qu'à mon avis, cela ne marche pas. Alors oui, bien entendu, vous allez pouvoir me citer deux ou trois exemples qui fonctionnent avec un mode de monétisation par la pub. Vous allez même pouvoir me dire que Facebook ou Google ne vivent que de la publicité. Mais Facebook et Google fonctionnent comme des régies publicitaires et c'est tout à fait différent. Quant aux rares projets qui arrivent à se financer tant bien que mal avec de la publicité, ce sont pour moi de parfaits exemples de coups de chance.

Pourquoi ? Je vais vous poser une question simple : vous, en tant que joueur ou utilisateur d'application, combien de fois avez-vous cliqué sur une publicité ? En ce qui me concerne, je n'ai jamais dû le faire. Mais par contre, combien de fois avez-vous pesté contre ces maudites publicités qui polluent votre espace visuel ? Ou trouvé qu'une telle publicité réduisait tellement votre expérience de jeu que cela en devenait problématique ? Personnellement, cela m'arrive assez régulièrement. Si vous avez répondu la

même chose que moi, pourquoi pensez-vous que les personnes qui joueront à votre jeu auront un comportement différent ? Voilà pourquoi je n'aime pas la monétisation par la publicité. Parce que pour moi, un moyen de monétisation avec des retombées plus qu'aléatoires et en plus, une réduction du plaisir de jouer, n'est pas un bon moyen de monétisation.

2. Certains joueurs paient

On va donc avoir ici des joueurs qui vont pouvoir jouer sans payer et d'autres qui vont choisir de payer. On va typiquement être dans des *business models* qui s'apparentent à ceux du *free to play*. Mais que faire payer à des joueurs qui peuvent jouer sans payer ?

2.1 Encourager ses joueurs à jouer à la Barbie

Les plus grosses ficelles sont celles qui marchent le mieux a-t-on coutume de dire. Et c'est assez souvent vrai. La monétisation par vente d'items supplémentaires fonctionne grâce à l'envie qu'ont les joueurs d'avoir l'avatar le plus beau possible (vous remarquerez que l'avatar peut aussi bien être ici un personnage humanoïde qu'une voiture). Pas seulement le plus beau, mais aussi le plus unique, le plus remarquable. Et un certain nombre sont prêts à payer pour cela, pour pouvoir acheter un plus joli chapeau que le personnage du voisin. Un exemple un peu vieux fut lorsque Blizzard, lors de je ne sais plus quelle occasion, mit en vente pour une durée limitée, une monture pour World of Warcraft, qui n'apportait rien à part un joli *character design*. Ils en vendirent des montagnes.

2.2 Proposer du contenu en avance

Pour vivre, votre jeu va avoir besoin de nouveautés, de nouveaux challenges, de nouveaux contenus. Vous pouvez tout à fait proposer en légère avance les nouvelles ressources à vos joueurs payants. Imaginons que vous ayez développé un jeu de Quiz. Vous avez défini que chaque semaine vous proposeriez un nouveau quiz. On pourrait imaginer que vos joueurs payants aient accès à ce nouveau quiz dès le lundi

matin, alors que les joueurs autres n'y auraient accès que le mercredi lorsque vous libérerez le fichier quiz. Alors oui, pendant trois jours, une petite partie des ressources de votre jeu ne sera pas libre. Cela pourra faire hurler certains puristes. Personnellement, je trouve que tout travail doit pouvoir mériter salaire et qu'un petit délai dans la libération d'une ressource n'est pas quelque chose qui doit vous amener au bûcher.

2.3 Proposer du contenu supplémentaire

C'est une modification du principe précédent, qui implique soit d'avoir une partie de vos ressources qui ne sont pas libres, soit d'ajouter des restrictions d'accès sur votre serveur de jeu. On voit ce système mis en place de manière de plus en plus régulière dans les MMORPG, mais c'est un vieux modèle de monétisation des jeux vidéo. Vous proposez du contenu qui n'est accessible qu'à vos joueurs payants. Cela peut être des zones géographiques supplémentaires à explorer si vous avez un jeu basé sur la découverte d'un monde, ou de nouvelles cartes de circuit si vous mettez en place un jeu de course de voitures, ou bien encore de nouveaux joueurs de basket ou l'accès à de nouveaux championnats si vous voulez développer un jeu de basket entre équipes inter-galactiques.

Cela peut être aussi des objets d'équipement pour votre avatar (là encore, que cela soit un personnage ou un objet). La différence avec le paragraphe précédent est que là, le contenu que vous proposez ne sera pas juste une amélioration du visuel de votre avatar, mais pourra avoir un vrai impact sur la « puissance » de votre personnage dans le jeu. Cela pourra être une épée magique super puissante ou un pare-choc ultra renforcé. Mais cela aidera vos joueurs à être meilleurs dans le jeu. Et là encore, être meilleur, voire le meilleur, c'est un moteur suffisamment important pour que les joueurs acceptent de passer à la caisse.

Ce type de vente de contenu, les objets qui augmentent la puissance, marche très bien avec des jeux « difficiles » qui vont demander un long entraînement ou des compétences aux joueurs. Ceux qui n'ont pas envie de s'entraîner ou qui ne sont pas assez bons pourront du coup faire jeu égal avec les joueurs meilleurs qu'eux.

2.4 Modifier le game design en faveur des joueurs qui paient

Votre jeu, comme tout jeu, repose sur des règles. Des règles qui limitent le joueur et encadrent ses possibilités. Vous pouvez par exemple définir que le joueur a un nombre de points d'action par jour et que chaque action qu'il fait va lui coûter un certain nombre de ses points d'action. Vous pouvez aussi définir qu'il peut faire autant d'actions qu'il veut, mais qu'une seule à la fois et que, malheureusement pour lui, une action prend du temps (un certain nombre d'heures) et que l'on ne peut pas mettre en attente des actions, qu'il faut attendre la fin d'une action pour en lancer une autre.... Bref, vous avez défini des règles.

Et vos joueurs, comme tous les joueurs, aimeraient bien pouvoir aller un peu au-delà de ces règles. Pour avoir un avantage sur les autres joueurs ou pour arriver plus vite à un niveau élevé du jeu. Vous pouvez monétiser cela, ces petites entorses aux règles.

Par exemple, si vous avez un jeu basé sur des points d'action, vous pouvez vendre des points d'action supplémentaires ou un accélérateur de récupération de points d'action (imaginons que vous regagnez un point d'action toutes les deux heures en temps normal, un joueur ayant acheté l'accélérateur en gagnerait un toutes les heures et demi). Si par contre vous avez un jeu basé sur des actions longues, vous pouvez soit vendre des accélérateurs de temps, soit proposer des aides de jeu payantes, comme par exemple une file d'attente d'actions qui permettra à vos joueurs de programmer des actions. (Un de mes amis était il y a quelques années fan d'un jeu de ce type-là ; pour ne pas avoir à payer, il lui arrivait assez régulièrement de mettre son réveil à trois heures du matin pour lancer une nouvelle action dès que l'action précédente était finie).

Mais vous pouvez aller encore plus loin, vous pouvez toucher à toutes les règles de votre jeu. Par exemple, réduire le brouillard de guerre des joueurs payants si vous utilisez cet effet dans votre jeu. Enfin, un système assez souvent mis en place, dans beaucoup de jeux pour téléphone d'ailleurs, est de permettre de jouer plus de fois. Pour tous les jeux plutôt *casual*, vous pouvez en effet

limiter le nombre de parties possibles par jour. Et permettre d'acheter des jetons supplémentaires. On en revient au bon vieux principe de la borne d'arcade et de son monnayeur.

2.5 Vendre des ressources

Un grand nombre de jeux nécessitent la collecte d'une ou plusieurs ressources pour pouvoir avancer. Cela peut être des dollars virtuels pour améliorer votre écurie de voitures de course, de l'acier pour construire des bâtiments et du plutonium pour les moteurs des vaisseaux spatiaux d'attaque de vos joueurs ou des dents de trolls des cavernes, mais il y a souvent cette ressource, surtout dans les jeux orientés gestion, que vos joueurs vont devoir passer des heures et des heures à collecter. Mais si certains de vos joueurs sont pressés, pourquoi ne pas leur proposer des packs de ressources ? Un petit coup de carte bleue et les voilà assis sur une montagne d'acier ou de dents de troll.

2.6 Proposer des services extérieurs à votre jeu

Imaginez que vous avez lancé un jeu de conquête spatiale. Chaque joueur gère une petite colonie et doit se débrouiller pour la faire grandir tout en combattant les pirates de l'espace et les autres joueurs. Il peut être vital pour un joueur de savoir quand il va se faire attaquer pour pouvoir préparer ses défenses. Vous pourriez donc proposer un service d'alerte par SMS.

Avec un jeu basé sur des parties « courtes » comme un Puissance 4, des courses ou des parties d'échecs, ce sont les slots de parties qui deviennent une ressource limitée. Vous pouvez alors proposer des abonnements qui permettront à vos joueurs d'être sûrs d'avoir toujours un circuit ou un plateau de jeu réservé quel que soit le moment où ils voudront jouer. Plus de file d'attente pour eux !

Ou alors, réutilisez l'un des modes de rémunération de Github à votre compte. Je m'explique. Github vous permet d'héberger gratuitement vos projets de développement logiciel, mais à une condition : il faut que le *repository* soit

accessible à tout le monde. Si vous voulez avoir un repository privé, il faut payer. Suivant le type de votre jeu, vous pouvez réutiliser ce principe. Dans le mode gratuit, les parties sont ouvertes à tous et n'importe qui peut se connecter dans votre partie. Par contre, le mode payant vous permet de définir des parties privées où seuls les utilisateurs que vous invitez peuvent se connecter.

3. Tous les joueurs paient

C'est le mode de monétisation le plus classique, qui nous vient directement du monde du jeu vidéo classique. Quelqu'un veut jouer à un jeu, il paie pour avoir accès soit à la version boîte, soit à un installateur dématérialisé. C'est un mode de fonctionnement, qui bien que compliqué à mettre en place, peut à mon avis fonctionner. Pour plusieurs raisons. Même si vous avez un jeu libre, sur certaines plateformes (comme les téléphones) il peut être très compliqué de le déployer soi-même. Quant à tous les jeux web, vu qu'après tout, le HTML 5 c'est surtout pour faire du Web, déployer sa propre instance de jeu où l'on sera seul à jouer n'a vraiment que peu d'intérêt. Après, il est vrai qu'il peut être difficile d'imaginer un jeu web qui fasse payer tous ses utilisateurs. Cela irait en effet encore plus loin dans la mouvance de dématérialisation, vos joueurs n'achèteraient même plus un fichier mais simplement un accès. Mais pourquoi pas ? Et puis, il semblerait que ce mode du tout payant puisse fonctionner. Très récemment, App.net a réussi à lever des fonds avec un système à la Kickstarter pour lancer un réseau social à accès payant. Alors pourquoi pas pour un jeu ?

Sans compter que l'on peut tenter de recycler les modes de vente utilisés par Radiohead ou les *Humble Indie Bundles* [2], à savoir permettre aux futurs joueurs de payer ce qu'ils pensent que le jeu vaut, en instaurant ou pas un prix d'achat minimum.

4. Les autres moyens de monétisation

Les précédents moyens de monétisation étaient classés en fonction de qui

paie. Pour ceux qui suivent, même s'il est fort probable que ce soit des joueurs qui paient, ce n'est pas une obligation, c'est donc pour cela qu'ils sont classés à part et pas du tout parce que je les considère moins rentables que les précédents.

4.1 Les dons

C'est tout bête et c'est le moyen le plus simple à mettre en place. C'est même encore plus simple que de mettre de la publicité. Il suffit de mettre un bouton vers votre interface de paiement préférée et d'attendre que de bonnes âmes cliquent dessus.

4.2 Le financement a posteriori de ressources

Imaginons que vous ayez développé un jeu de courses de voitures. Vous voulez mettre en place un nouveau modèle de voiture. Vous pouvez très bien procéder de la façon suivante. Vous budgétisez combien va vous coûter la création de ce modèle graphique. Et ensuite, vous mettez en place une souscription le concernant. Vous avez ensuite plusieurs moyens de procéder. Soit vous attendez que le budget soit atteint avant de commencer la création de la ressource. Soit vous la créez tout de suite, mais sans la libérer. Vous ne permettez ensuite qu'aux personnes ayant participé à la souscription de l'utiliser dans le jeu. Et ce n'est qu'une fois le budget atteint que la ressource en question devient libre et que tout le monde peut l'utiliser.

4.3 Le Crowdfunding de la totalité du projet

Depuis quelques temps maintenant, les plateformes de *crowdfunding* ont le vent en poupe. Il y a bien entendu la plus connue, Kickstarter [3]. Mais ce n'est pas le seul et rien qu'en France, on pourra citer Ulule [4] ou KissKissBankBank [5] qui ont tous les deux permis à un nombre important de projets d'exister, dont des projets de jeux vidéo.

Je ne vais pas trop m'étendre sur le mode de fonctionnement du Crowdfunding. Juste brièvement rappeler que le principe en est simple : vous demandez aux internautes de croire en votre projet et de le financer. Vous devez définir un budget minimum à atteindre, différents paliers de financement (par exemple cinq, dix,

et cinquante euros) et à quoi donnent accès ces paliers. Par exemple, pour cinq euros, un accès bêta au jeu et pour mille euros un repas avec vous. Ensuite, il n'y a plus qu'à attendre. La plupart des plateformes de ce style fonctionnent sur un délai de trente jours. Si à la fin du délai, vous avez récolté suffisamment de promesses de financement, celles-ci se transforment en financement réel et vous n'avez plus qu'à vous mettre au travail. Dans le cas contraire, les internautes récupèrent leur pécule et vous n'avez plus qu'à trouver un autre moyen de financement.

4.4 Pourcentage sur les transactions

Bien que ne l'ayant vu que dans deux jeux à ce jour, à savoir Second Life et Diablo 3 et doutant très fortement de la possibilité de le mettre en place dans un jeu qui ne soit pas un AAA attirant des centaines de milliers de joueurs, je ne pouvais pas ne pas parler de cette possibilité pour les concepteurs d'un jeu de permettre des transactions en euros entre joueurs ; pour par exemple revendre des objets que l'on aurait trouvé sur le cadavre des monstres ; et de percevoir un pourcentage sur ces transactions. Voilà, je le cite pour tenter d'être exhaustif, mais tenter un tel moyen de monétisation pour un jeu libre c'est à mon avis aller droit dans le mur. Je vous aurais prévenu.

5. Conclusion

Lors de cet article, nous avons fait un tour rapide de quelques-unes des possibilités de monétisation de votre futur jeu libre. Ce n'est toutefois pas une liste exhaustive des différentes façons de vous rémunérer pour le travail que vous avez fourni. Ne vous bridez pas si vous trouvez d'autres moyens de monétisation qui fonctionnent. N'hésitez d'ailleurs pas à m'envoyer un mail pour partager vos trouvailles, si le cœur vous en dit.

Dans cette conclusion, je voulais revenir sur un aspect important des choses. Bien souvent, quand je parle de monétisation de jeux libres, on m'oppose le fait que le jeu étant libre, les joueurs peuvent très bien y jouer librement en récupérant les sources et les ressources de celui-ci et en l'installant sur leurs machines, ou lorsque l'on parle d'un jeu web, en réseau

en déployant une instance sur un serveur. C'est tout à fait vrai.

Mais j'ai toujours trouvé ennuyeux de jouer tout seul à un jeu à highscore ou à un jeu prévu pour du multijoueur ou de la compétition entre les joueurs. Jouer contre soi-même, à la longue, c'est fatigant. Alors autant il est vrai que pour des jeux vidéo classiques la monétisation peut être compliquée, autant pour du jeu vidéo web - et il ne faut pas se leurrer, les jeux en HTML 5 sont pour l'instant dans leur immense majorité des jeux web ou smartphone - elle me semble beaucoup plus facile.

Pour une raison toute simple : un jeu pour fonctionner a besoin d'une masse critique. Cela se comprend tout à fait pour un jeu multijoueur ou un jeu à highscore. Il faut un certain nombre de joueurs pour que la compétition soit intéressante. Mais c'est aussi le cas pour un jeu monojoueur. Je m'explique. Imaginons un jeu de résolution d'éénigme policière en *point and click*. Prenons comme hypothèse que le nombre de joueurs n'influe en aucun cas sur le plaisir de jeu d'un joueur. Mais si votre jeu n'a que peu de joueurs, vous serez moins enclin à ajouter du contenu, ici de nouvelles énigmes policières, et donc, votre jeu périclitera petit à petit parce que vous, en temps que développeur de logiciel libre, vous trouverez mieux à faire de votre temps. Il vous faut donc bien une masse critique pour que votre jeu fonctionne.

Pourquoi cela rend-il potentiellement plus facile la monétisation d'un jeu web ? Parce qu'atteindre cette masse critique, ce n'est pas trivial, cela demande un investissement. Investissement qui fera office de barrière à l'entrée pour ceux qui auront envie de déployer votre jeu sur leur serveur pour y jouer et qui donc viendront jouer sur votre instance de jeu. Et s'il n'y a que votre instance de jeu qui est amusante à jouer, parce que les autres instances de votre jeu n'ont pas atteint la masse critique, cela vous laisse plus de liberté pour déployer des moyens de monétisation.

Pour finir, une dernière chose. Si l'on reprend les différents types de monétisation vus dans cet article et qu'on les regarde à travers le prisme de l'actualité, on peut se dire que ce qui a le vent en poupe actuellement, c'est de faire payer une partie des joueurs. De nombreux MMORPG fonctionnent maintenant

avec ce principe-là pour rémunérer, ainsi que de nombreux jeux casual web ou sur smartphone. Or, on sait bien que seule une petite fraction des joueurs deviendra des joueurs payants. Il vous faut donc là aussi une masse critique. Le paradoxe c'est que mal utilisée, la vente d'objets supplémentaires, d'options puissantes, ou en un mot de priviléges, peut faire fuir les joueurs qui se contentent du jeu en mode gratuit. Et si ces joueurs-là partent vers d'autres contrées, vos joueurs payants les suivront vu que votre jeu n'aura plus la masse critique nécessaire pour fonctionner.

J'ai un exemple parfait pour ce problème. Il y a quelques années, en 2004, je jouais à un jeu de « duel ». Le jeu en lui-même était très simple. On avait un personnage avec une classe, des armes, des sortilèges et un équipement. On le faisait combattre contre d'autres personnages. Le combat était résolu automatiquement par le moteur de jeu. Le jeu tournait bien jusqu'à ce que le concepteur décide d'ajouter des armes et des sortilèges payants. Il devint alors impossible de battre le personnage d'un joueur payant, les armes payantes étant mille fois plus puissantes que celles classiques. En moins de deux mois, le jeu perdit la totalité de ses joueurs. Donc, si vous choisissez ce système de monétisation, il faut bien que vous soyez conscient que l'équilibrage entre les possibilités des joueurs qui jouent en mode gratuit et ceux qui sont en mode payant est primordial. Parce que ce sont les joueurs qui paient qui vous apporteront de l'argent, mais ce sont les joueurs qui ne paient pas qui vous apporteront les joueurs qui paient et qui feront que votre jeu vit ou pas. ■

Références

- [1] Rencontres Mondiales du Logiciel Libre : <http://2012.rmll.info/>
- [2] The Humble Bundle : www.humblebundle.com
- [3] Kickstarter : <http://www.kickstarter.com/>
- [4] Ulule : <http://fr.ulule.com/> (et sa section jeux : <http://fr.ulule.com/#!tags/games/>)
- [5] KissKissBankBank : <http://www.kisskissbankbank.com> (et sa section jeux vidéo : <http://www.kisskissbankbank.com/fr/discover/categories/jeux-video>)



ET SI ON JOUAIT CARTES SUR TABLE ?

par Jean-Michel Armand

Un jeu vidéo implique souvent une représentation spatiale des choses. Pour dire les choses simplement, un jeu vidéo implique souvent la mise en place d'un plateau de jeu. Dans cet article, nous allons voir quelles sont les caractéristiques d'un plateau de jeu, ainsi que les différentes manières de modéliser les choses.



1. Un plateau de jeu, c'est comme un plateau de fromages, mais sans le fromage

Des plateaux de jeu, il y en a presque autant qu'il y a de jeux. D'ailleurs, il serait plus juste de parler d'espace de jeu et non de plateau de jeu. Nous allons voir quelques-unes des caractéristiques des espaces de jeu courants.

1.1 Discrets ou continus

Les espaces de jeu peuvent être de deux types, soit discrets, soit continus. Par discret, on entend un plateau de jeu où l'une au moins de ces propriétés est vraie :

- ses points/cases/zones sont séparés les uns des autres ;
- le nombre de positions que pourra avoir un objet du jeu est dénombrable.

En fait, l'utilisation de cases dans la modélisation d'une position dans un jeu entraîne bien souvent le fait que le plateau de jeu soit de type discret. Un bon exemple de plateau discret est le plateau d'un jeu de dames. Le plateau d'un jeu de dames possède en effet cent cases et pas une de plus. Les seules positions possibles d'un pion sont donc sur l'une des cent cases du damier (ou hors du plateau de jeu s'il a été mangé).

Un espace continu est lui un espace « sans trou », avec un nombre indénombable de positions possibles pour les objets qui s'y trouvent. Un exemple d'espace continu dans un jeu vidéo pourrait être l'espace intersidéral dans Eve Online. Les vaisseaux des joueurs et des personnages non joueurs n'ont pas de position se limitant à des cases. Il va s'en dire qu'il peut devenir plus difficile de modéliser des espaces continus. Une astuce possible est donc d'utiliser des espaces discrets en les faisant passer pour continus. C'est typiquement ce que l'on avait dans certains vieux jeux en 3D. Intellectuellement, on pourrait penser que l'espace de jeu d'un jeu en 3D, où l'on déplace un avatar, devrait être continu. Mais suivant les simplifications que l'on se permet de faire au niveau de son jeu, on peut utiliser un espace discret pour modéliser un espace continu, en le camouflant le plus possible.

1.2 Avec des dimensions

Un espace de jeu possède un certain nombre de dimensions. Dans l'exemple du jeu de dames, on a un espace à deux dimensions. Le monde d'Eve Online est lui un espace à trois dimensions. Suivant les cas, le nombre de dimensions d'un espace de jeu peut d'ailleurs changer en fonction de ce que vous voulez modéliser. Imaginons que vous voulez développer un jeu de baby-foot. Vous pourriez imaginer

d'utiliser une modélisation bidimensionnelle continue. Mais suivant comment l'on joue, il est possible de faire sauter la bille, ce qui transforme votre espace bidimensionnel en espace tridimensionnel. Est-ce qu'il existe d'autres espaces de jeu ? À une ou zéro dimension ? On pourrait dire que oui. Un jeu uniquement mental, par exemple le jeu des dix questions (un joueur doit deviner à quoi pense l'autre en posant dix questions) est un jeu ayant un espace de jeu à zéro dimension. De même, un jeu de type jeu de l'oie est un jeu à une dimension.

1.3 La notion de données secrètes

Une notion importante de votre espace de jeu va être celle du secret. Quelles sont les informations que vos joueurs connaîtront et quelles seront les règles qui leur permettront d'en apprendre plus ? Les échecs sont par exemple un jeu où le secret n'existe pas. Chaque joueur voit totalement le plateau de jeu, ainsi que la position de chacune des pièces. A contrario, la bataille navale est un jeu où vous ne savez rien du tout de l'autre joueur et où vous apprendrez petit à petit, en tâtonnant, où sont ses bateaux. Dans les jeux RTS, c'est un peu le même principe qui est mis en place avec le brouillard de guerre qui cache les zones du plateau qui sont trop éloignées de vos différentes unités pour être vues.

RTS, kesako ?

RTS est l'acronyme de *Real Time Strategy* (STR en français). Ce sont des jeux de stratégie qui mixent deux types de gestion, la gestion des ressources et la gestion des combats. La gestion des ressources consiste à gérer la construction de sa base, de ses unités, la récupération des ressources ou alors à simplement gérer l'arrivée des renforts. La gestion des combats est bien souvent du type micro-gestion, c'est-à-dire que l'on va gérer chaque unité à un niveau individuel pour essayer d'obtenir un avantage au niveau général du combat. Certains RTS qui proposent des batailles avec un nombre énorme d'unités ne proposent pas de micro-gestion.

La grande différence entre les RTS et les jeux de stratégie traditionnels tels que les wargames provient du caractère temps réel du jeu. Il ne suffit plus en effet de réfléchir et d'être un bon stratège, il faut aussi avoir une grande réactivité et parfois même d'excellents réflexes. Cette nécessité de réagir rapidement, d'une manière presque automatique, est parfois pointée comme un travers des RTS du fait de la possibilité qu'elle entraîne une diminution des compétences stratégiques nécessaires pour qu'un joueur gagne.

Historiquement, on s'accorde à dire que l'un des premiers RTS est Dune 2, sorti en 1992 et développé par Westwood Studios.

1.4 Espaces imbriqués

Il peut être intéressant pour des simplifications de modélisation et de game design de modéliser un espace de jeu comme un ensemble d'espaces de jeu imbriqués les uns dans les autres. Les joueurs qui sont dans l'un des sous-espaces se retrouvent cloisonnés à ce sous-espace et n'ont plus de visibilité vers l'espace « parent ». C'est le cas dans un grand nombre de jeux de rôle ou de *hack and slash*, où lorsqu'un joueur entre dans un bâtiment, il ne voit plus du tout ce qu'il peut se passer autour du bâtiment. Ce petit effet de modélisation

permet de simplifier grandement les modélisations et correspond en plus assez souvent à nos propres modèles de schématisation des choses.

2. Modélisation des espaces de jeu et de leurs propriétés

Je ne traiterai pas ici des modélisations des espaces de jeu à zéro ou une dimension qui sont des cas très particuliers et que l'on ne retrouve que peu souvent dans les jeux vidéo.

2.1 Cas d'un espace discret : modélisation par matrice de cases et tiles

Un vieux système de modélisation d'espace de jeu est le système de modélisation par *tiles*. Le principe en est très simple. On découpe le plateau de jeu en cases de petite taille. Les cases sont de forme géométrique simple, le plus souvent carrées. Mais on peut avoir des tiles hexagonales ou en losange. On pourrait en fait utiliser n'importe quelle forme géométrique. Chacune des cases sera ensuite pavée avec un petit élément de décor. On modélise le niveau par une matrice bidimensionnelle, dont chaque entrée contient les caractéristiques d'une case (dont l'image à utiliser). Ensuite, on rassemble toutes les petites images de décor (les tiles) dans un seul et unique fichier que l'on nomme un *tile set* (qui du coup, va énormément ressembler à un fichier de ressources d'un site web utilisant la technique des sprites CSS).

Cette technique a de nombreux avantages. On peut tout d'abord modéliser ainsi de très grands niveaux à peu de coût au niveau des ressources. Ensuite, la mise en place des fichiers de cartes est très simple. Il suffit d'utiliser de simples fichiers textes, que l'on pourra même compresser de la manière dont on le souhaite.

Concernant les éléments de décor qui pourraient faire plus d'une tile de

taille, il suffit de les découper en un ensemble de tiles. On peut même, dans certains cas, utiliser ce mécanisme dans des jeux 3D. Concernant les collisions entre les objets, là aussi cela reste très simple. Un objet - que cela soit l'avatar d'un joueur, un tank ou la maison du personnage non joueur qui va vous donner votre prochaine quête - remplit une ou plusieurs cases. Lorsque l'on veut savoir si un objet entre en collision avec un autre, il suffit de tester si les cases qu'ils occupent sont les mêmes.

Le problème d'une modélisation par tiles est la répétitivité des décors qu'elle entraîne. Ce problème peut être contré par une méthode toute simple et qui est rendue possible avec la puissance actuelle de nos machines : ne plus utiliser de tiles répétitives. Continuer à pavier l'espace de jeu grâce à des plateaux constitués uniquement de formes géométriques simples, mais pour ce qui est des décors, utiliser des décors « en un seul tenant » et non répétitifs. Cela consomme effectivement plus de ressources et cela demande sans aucun doute possible un travail bien plus énorme en ce qui concerne les graphismes, mais cela permet de résoudre totalement le problème de la répétitivité des décors et donc, de protéger vos joueurs de l'ennui, en tout cas en ce qui concerne les décors.

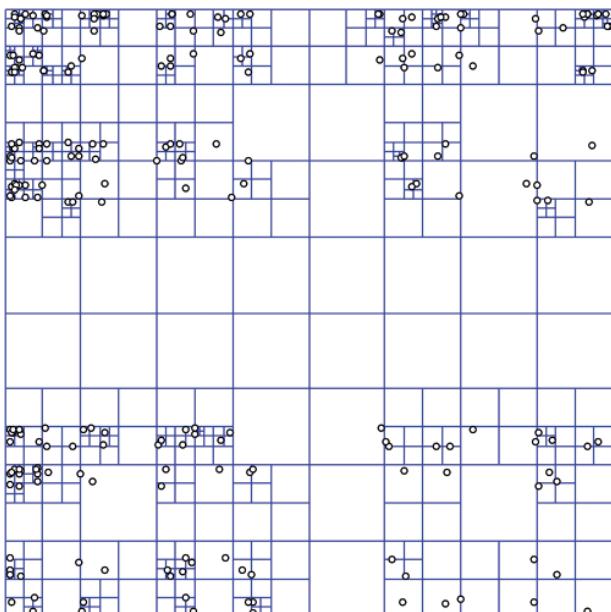
2.2 Cas d'un espace continu, coordonnées et vitesse

La méthode précédente avait l'avantage d'être puissante et simple à mettre en place. Mais elle ne permet de modéliser que les espaces de jeu qui sont discrets ou que l'on peut modéliser de façon discrète tout en faisant croire au joueur qu'il est dans un espace continu. La modélisation d'un espace continu dans lequel on pourra trouver à la fois des objets immobiles et des objets mouvants est légèrement plus difficile que la précédente. Elle va faire appel à quelques notions mathématiques de l'époque où vous étiez au lycée.

En effet, pour modéliser un tel espace continu, il va vous falloir vous appuyer sur la modélisation géométrique des positions. C'est-à-dire utiliser un repère orthonormé (x,y,z) et définir pour chaque objet une position dans l'espace (grâce à ses coordonnées cartésiennes Xm,Ym,Zm), ainsi que son vecteur vitesse et, si votre jeu le prend en charge, son accélération. Ensuite, à chaque mouvement de vos objets, il suffira de leur appliquer leur vecteur vitesse, modifié au préalable grâce à leur accélération.

Arrivé à ce point de l'explication, vous vous rendez bien compte qu'il manque quelque chose. En effet, un objet, comme par exemple l'avatar de votre héros pourchassé par des extraterrestres affamés, n'est pas constitué d'une coordonnée. Il est même représenté par un avatar graphique à deux ou trois dimensions. En fait, les coordonnées dont je parle ne sont que les coordonnées du point d'accroche de l'objet dans l'espace de jeu. Nous avons donc maintenant une méthode pour modéliser et faire se déplacer des objets dans un espace continu. Mais comment gérer les collisions dans un tel espace ?

Gérer une collision parfaite, modèle avec modèle, devient bien vite illusoire dès qu'il y a beaucoup de modèles et que ceux-ci ne sont pas de simples formes géométriques. C'est à ce stade qu'apparaissent les boîtes de collisions. Une boîte de collisions est une forme géométrique simple, très souvent un parallélépipède, qui va contenir vos objets de la manière la plus serrée possible. Lorsque l'on voudra tester si deux objets rentrent en collision, il suffira de faire le test entre leurs boîtes de collisions respectives. Un objet peut d'ailleurs avoir plusieurs boîtes de collisions. Si par exemple vous mettez en place une règle de type brouillard de guerre, les unités de vos joueurs pourraient avoir deux boîtes de collisions : celle pour gérer les collisions réelles avec les éléments du décor et celle permettant de définir le volume dans lequel le brouillard de guerre n'agirait pas.



*Fig. 1 : Un quadtree, par David Eppstein
(source : Wikipédia)*

Il semblerait que l'on ait résolu tous les problèmes. Il en reste encore un, même en utilisant des boîtes de collisions, tester les collisions entre tous les objets de votre espace serait tout d'abord inutile et ensuite, potentiellement bien trop coûteux en ressources. On utilise donc des techniques de partitionnement de l'espace pour réduire rapidement le nombre d'objets qui ont une probabilité de rentrer en collision entre eux. On utilise pour cela, lorsque l'on est en 2D, un *quadtree* et lorsque l'on est en 3D un *octree*.

Quadtree

Un quadtree est une structure de données arborescente qui permet de partitionner efficacement un espace en deux dimensions. Chaque nœud peut avoir quatre fils, qui peuvent eux-mêmes se diviser récursivement. On subdivise ainsi chaque espace modélisé par un nœud en quatre sous-espaces plus petits. Le pendant en trois dimensions des quadtrees sont les octrees, qui eux, se baseront sur des nœuds pouvant se diviser en huit.

3. Conclusion

On l'a vu, il y a de nombreuses façons de penser et de modéliser ses espaces de jeu. De plus, il n'y a pas de façon gravée dans le marbre pour modéliser un espace de jeu donné. En fait, le choix des attributs d'un espace de jeu et de sa méthode de modélisation dépend bien souvent des règles que vous voulez mettre en place dans votre jeu. Il peut aussi être nécessaire d'avoir plusieurs modélisations différentes pour un espace de jeu.

Par exemple, imaginons que vous vouliez développer un jeu de bataille de tanks. On pourrait imaginer d'avoir une première modélisation bidimensionnelle et continue pour gérer les déplacements des tanks sur le terrain et les collisions entre eux, et une modélisation tridimensionnelle continue en ce qui concerne les déplacements des obus.

Ce qui est important, en dehors du fait d'utiliser les bonnes techniques de développement informatique pour modéliser les choses, c'est lorsque vous voulez réfléchir sur vos espaces de jeu et leurs interactions avec les choses contrôlées par le joueur, de retirer tous les éléments esthétiques ou graphiques de vos espaces de jeu pour ne garder que ce qui est purement fonctionnel dans celui-ci. Vous verrez alors que vous pourrez grandement simplifier les choses.

Pour finir, un exemple de différence entre l'espace de jeu vu par le joueur et l'espace de jeu fonctionnel, mis en place au niveau du code, dans Doom 2, l'un des FPS les plus connus qui soit. On pourrait penser que comme notre personnage se déplace dans un univers 3D, la modélisation de celui-ci est un espace 3D. Mais ce n'était pas le cas, les niveaux de jeu de Doom étaient représentés sur un espace bidimensionnel, les informations d'élévation étant stockées séparément. Cela interdisait à Doom d'avoir des pièces au-dessus d'autres pièces, mais c'est, en autres, cette astuce qui a permis à John Carmack de faire tourner le jeu sur les ordinateurs de l'époque. ■

Abonnez-vous !

Profitez de nos offres d'abonnement spéciales disponibles au verso !

Économisez plus de

20%*

* Sur le prix de vente unitaire France Métropolitaine

Numéros de
6 Linux Pratique

Téléphonez au
03 67 10 00 20
ou commandez
par le Web

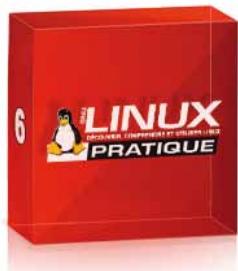
Les 3 bonnes raisons de vous abonner :

- Ne manquez plus aucun numéro.
- Recevez Linux Pratique dès sa parution chez vous ou dans votre entreprise.
- Économisez 9,00 €/an !

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur www.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

par ABONNEMENT :



30€*

au lieu de 39,00 €* en kiosque

Économie : 9,00 €*

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITaine
Pour les tarifs hors France Métropolitaine, consultez notre site :
www.ed-diamond.com

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>



Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

Voici mes coordonnées postales :

Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : www.ed-diamond.com/cgv et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

PROFITEZ DE NOS OFFRES D'ABONNEMENT SPÉCIALES POUR LIRE PLUS ET FAIRE DES ÉCONOMIES !

► Voici nos offres de couplage

1	Linux Pratique (6 nos) par ABO : 30€*  au lieu de 39,00€** en kiosque Economie : 9,00 €	2	Linux Pratique Essentiel (6 nos) + Linux Pratique (6 nos)  par ABO : 57€* au lieu de 78,00€** en kiosque Economie : 21,00 €	3	GNU/Linux Magazine (11 nos) + Linux Pratique (6 nos)  par ABO : 78€* au lieu de 121,50€** en kiosque Economie : 43,50 €	4	GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos)  par ABO : 83€* au lieu de 130,50€** en kiosque Economie : 47,50 €		
5	+ GNU/Linux Magazine (11 nos) + Misc (6 nos)  par ABO : 84€* au lieu de 133,50€** en kiosque Economie : 49,50 €	6	+ GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos)  par ABO : 110€* au lieu de 169,50€** en kiosque Economie : 59,50 €	7	+ GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Misc (6 nos)  par ABO : 116€* au lieu de 181,50€** en kiosque Economie : 65,50 €	8	+ GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Misc (6 nos)  par ABO : 143€* au lieu de 220,50€** en kiosque Economie : 77,50 €	9	Linux Pratique Essentiel (6 nos) + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Misc (6 nos)  par ABO : 173€* au lieu de 259,50€** en kiosque Economie : 86,50 €
10	MISC (6 nos) + MISC Hors-Série (2 nos)  par ABO : 44€* au lieu de 69,00€** en kiosque Economie : 25,00 €	11	Linux Pratique (6 nos) + Linux Pratique HS (3 nos)  par ABO : 42€* au lieu de 63,00€** en kiosque Economie : 21,00 €	12	Linux Pratique Essential (6 nos) + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos) + Misc (6 nos) + MISC Hors-Série (2 nos)  par ABO : 199€* au lieu de 301,50€** en kiosque Economie : 102,50 €		15	Linux Pratique Essential (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos)  par ABO : 72€* au lieu de 102,00€** en kiosque Economie : 30,00 €	

Vous pouvez également vous abonner sur : www.ed-diamond.com ou par Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

Nos Tarifs s'entendent TTC et en euros	F	D	T	E1	E2	EUC	A	RM
	France Métro	DOM	TOM	Europe 1	Europe 2	Etats-Unis Canada	Afrique	Reste du Monde
1 Abonnement LP	30 €	33 €	36 €	37 €	36 €	38 €	37 €	41 €
2 LPE + LP	57 €	62 €	69 €	71 €	69 €	73 €	71 €	79 €
3 GLMF + LP	78 €	85 €	96 €	99 €	95 €	101 €	98 €	111 €
4 GLMF + GLMF HS	83 €	89 €	101 €	104 €	100 €	105 €	103 €	116 €
5 GLMF + MISC	84 €	90 €	102 €	105 €	101 €	107 €	104 €	117 €
6 GLMF + GLMF HS + Linux Pratique	110 €	119 €	134 €	138 €	133 €	140 €	137 €	154 €
7 GLMF + GLMF HS + MISC	116 €	124 €	140 €	144 €	139 €	146 €	143 €	160 €
8 GLMF + GLMF HS + MISC + LP	143 €	154 €	173 €	178 €	172 €	181 €	177 €	198 €
9 GLMF + GLMF HS + MISC + LP + LPE	173 €	186 €	209 €	215 €	208 €	219 €	214 €	239 €
10 MISC + MISC HS	44 €	47 €	53 €	55 €	52 €	56 €	54 €	60 €
11 LP + LP HS	42 €	46 €	52 €	54 €	51 €	55 €	53 €	60 €
12 GLMF + GLMF HS + MISC + MISC HS + LP + LP HS + LPE	199 €	214 €	242 €	250 €	239 €	254 €	247 €	277 €
15 LPE + LP + LP HS	72 €	78 €	88 €	91 €	87 €	93 €	90 €	101 €

* Europe 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède

* Europe 2 : Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande

* Toutes les offres d'abonnement : en exemple, les tarifs ci-dessus correspondant à la zone France Métro (F) ** Base tarifs kiosque zone France Métro (F)

* Zone Reste du Monde : Autre Amérique, Asie, Océanie

* Zone Afrique : Europe de l'Est, Proche et Moyen-Orient

Mes choix :

Mon 1er choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 2ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 3ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Je sélectionne ma zone géographique (F à RM) :		
J'indique la somme due : (Total)		€

Exemple : je souhaite m'abonner à l'offre GNU/Linux Magazine + GNU/Linux Magazine Hors-série + MISC (offre 7) et je vis en Belgique (E1), ma référence est donc 7E1 et le montant de l'abonnement est de 144 euros.

Je choisis de régler par :

Chèque bancaire ou postal à l'ordre des Éditions Diamond

Carte bancaire n°

Expire le :

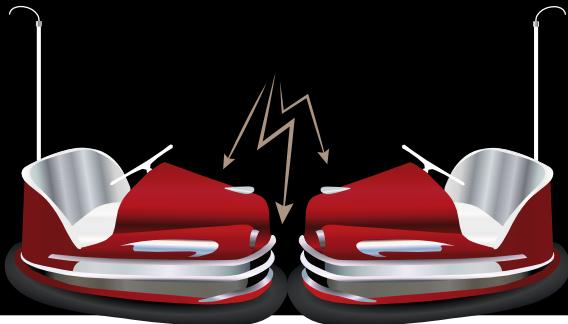
Cryptogramme visuel :

Date et signature obligatoire



DÉTECTION DES COLLISIONS

par Tristan Colombo



Tous les jeux n'ont pas forcément besoin de détecter les collisions entre différents objets, mais si vous avez besoin de vérifier qu'un personnage ne rencontre pas un ennemi ou ne traverse pas un mur, il faudra effectuer quelques tests.



Il n'existe pas une unique façon de détecter les collisions entre objets. De nombreux algorithmes peuvent être mis en place, il faut simplement choisir le plus adéquat à la situation que vous souhaitez traiter. Dans cet article, nous ne traiterons que de jeux en deux dimensions, la détection de collisions en trois dimensions étant plus complexe. Nous explorerons plusieurs solutions qui se prêteront chacune à des cas particuliers. Une petite précision avant de commencer : j'espère que vous n'êtes pas allergique aux mathématiques en général et à la géométrie en particulier...

1. Détection par boîtes

Il s'agit du mode de détection de collision le plus simple à mettre en place. Tous les objets du jeu sont bordés par un rectangle dont les coordonnées sont forcément connues comme le montre la figure 1.

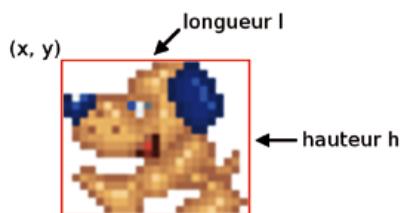


Fig. 1 : Sprite positionné en (x, y) et ayant une taille de $l \times h$ pixels

Imaginons que notre personnage, le chien de la figure 1, doive récupérer des sacs d'argent tombant du ciel. Nous allons devoir tester plusieurs situations de collisions, comme le montre la figure 2. Les segments de chaque boîte encadrant les sprites ont été nommés pour pouvoir énoncer plus simplement les tests à effectuer : Ch pour Chien haut, Cb pour Chien bas, Cg pour Chien gauche et Cd pour Chien droit, Ob pour Or bas, Oh pour Or haut, Og pour Or gauche et Od pour Or droit. Les collisions possibles sont :

- Ch avec Ob : le sac tombe sur la tête du chien ;
- Cg avec Od : le chien se déplace vers la gauche alors que le sac n'a pas encore touché le sol sur sa gauche ;
- Cd avec Og : le chien recule vers la droite alors que le sac n'a pas encore touché le sol sur sa droite.

Il y aura quatre tests à effectuer pour savoir si la collision a eu lieu. (xc, yc) étant les coordonnées du coin supérieur gauche du sprite du chien de taille $lc \times hc$ et (xo, yo) étant les coordonnées du sprite du sac de taille $lo \times ho$, les calculs seront :

- $yc > (yo + ho)$ et $(yc + hc) < yo$: permet de s'assurer que le chien et le sac sont à des hauteurs compatibles avec une collision. Par exemple, si le sac se trouve juste au-dessus du chien, c'est l'information de hauteur qui permettra de dire que le chien n'a pas attrapé le sac.
- $xc < (xo + lo)$ et $(xc + 1) > xo$: permet de s'assurer que les deux objets sont bien alignés.

La figure 3 (page suivante) illustre ces différents tests : les segments bleus représentent les bordures du sprite du sac, les segments jaunes représentent les bordures du sprite du chien et les zones vertes indiquent un recouvrement des segments.

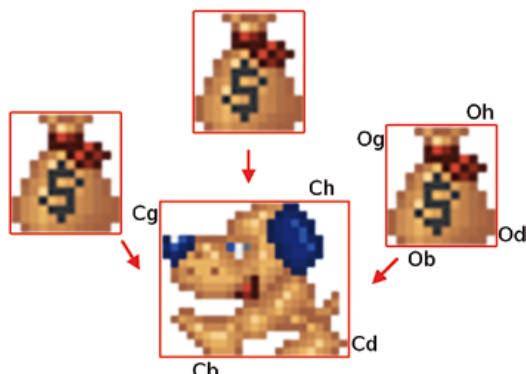


Fig. 2 : Exemples de collisions possibles entre deux sprites

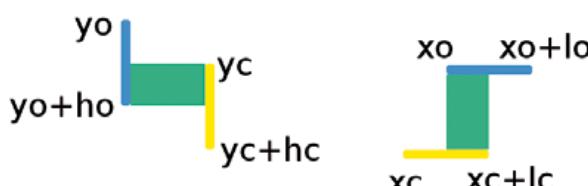


Fig. 3 : Collision entre deux objets par le haut

Cette méthode de détection est peu gourmande en ressources. Par contre, elle n'est pas très précise : une collision peut être détectée alors qu'aucun pixel des deux sprites n'est rentré en contact (Fig. 4). Si vous avez besoin d'être très précis dans vos détections, il faudra adapter l'algorithme précédent... au détriment de la vitesse.

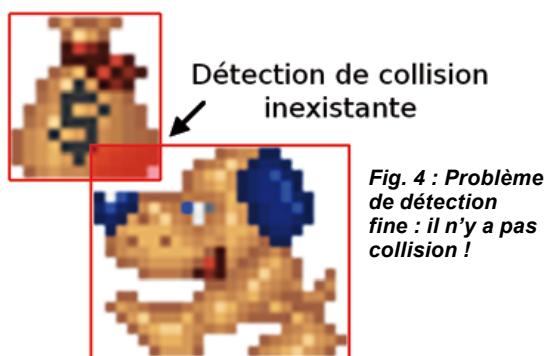


Fig. 4 : Problème de détection fine : il n'y a pas collision !

1.1 Détection fine par boîtes

La première solution consiste non plus à border vos objets par une unique boîte, mais à leur assigner un ensemble de boîtes déterminant leur contour, comme le montre la figure 5.

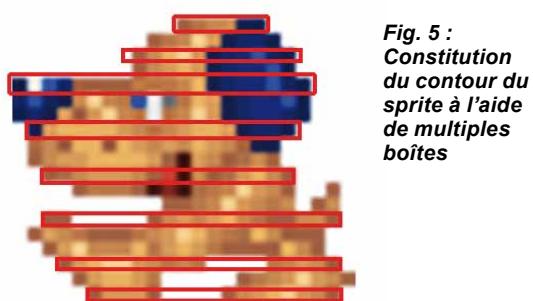


Fig. 5 : Constitution du contour du sprite à l'aide de multiples boîtes

Plus vous ajouterez de boîtes, plus la détection sera fine et plus le nombre de calculs (similaires à ceux de l'algorithme basique) sera élevé. Il faudra trouver un compromis permettant de spécifier les contours du sprite moins vaguement qu'avec une seule boîte, tout en évitant d'utiliser une boîte par ligne de pixels.

1.2 Détection fine par pixels

La seconde solution consiste à déterminer des points de collision, sortes de points de contrôle indiquant de façon

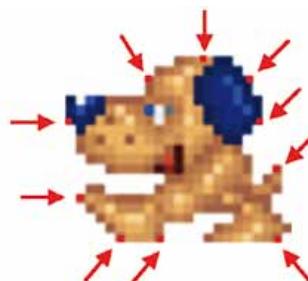


Fig. 6 : Utilisation de points de collision pour détecter les contacts

très vague le contour du sprite. La figure 6 montre le sprite du chien sur lequel nous avons ajouté 10 points de collision.

L'algorithme consistera ici à tester pour chacun des points si le pixel d'arrivée (après le déplacement) est libre ou non. Là encore, il faudra éviter d'utiliser trop de points de collision au risque de ralentir considérablement le jeu.

2. Détection par cercles

Il existe une méthode de détection qui est très proche de la méthode de détection par boîtes : la détection par cercles. Au lieu de délimiter les contours des sprites par des boîtes, nous utiliserons ici des cercles (Fig. 7).



Fig. 7 : Détermination du contour d'un sprite par un cercle

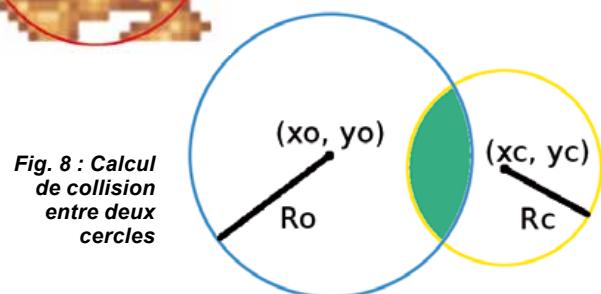


Fig. 8 : Calcul de collision entre deux cercles

Pour savoir si deux cercles se touchent ou s'il y a intersection comme le montre la figure 8, il faut calculer la distance entre leurs centres et la comparer avec la somme de leurs rayons :

- si la distance est supérieure à la somme des rayons, les cercles ne se touchent pas ;
- si la distance est égale à la somme des rayons, les cercles se touchent ;
- enfin, si la distance est supérieure à la somme des rayons, il y a intersection entre les deux cercles.

Le calcul de la distance entre deux points s'obtient grâce au théorème de Pythagore (sqrt représente la racine carrée) :

$$d = \text{sqrt}((xo - xc)^2 + (yo - yc)^2)$$

Donc si $d \leq (Ro + Rc)$, alors il y a collision.

3. Collision de lignes

Dans certains jeux, il peut arriver que des éléments soient représentés par des lignes (ou pour être plus précis, des segments). Pour déterminer la position du segment, nous connaissons forcément les coordonnées des points de ses extrémités : (x_a, y_a) et (x_b, y_b) . Or, l'équation d'une droite est du type $y = ax + b$. Donc, pour connaître l'équation de ce segment, il suffit de poser le système suivant :

$$y_a = ax_a + b \text{ et } y_b = ax_b + b$$

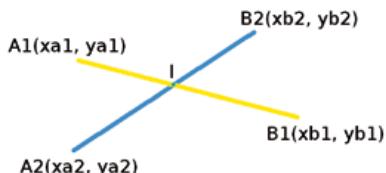


Fig. 9 : Calcul de collision entre deux lignes

On obtient alors :

$$a = (yb - ya) / (xb - xa) \text{ et } b = (ya(xb - xa) - xa(yb - ya)) / (xb - xa)$$

Donc :

$$y = ((yb - ya) / (xb - xa))x + (ya(xb - xa) - xa(yb - ya)) / (xb - xa)$$

avec $|xa| < |x| < |xb|$.

Pour savoir si deux lignes rentrent en collision, il faudra tester si elles possèdent un point commun. Les coordonnées de ce point permettront forcément de résoudre à la fois l'équation de la première et de la seconde ligne (Fig. 9).

Pour deux segments $[A1, B1]$ et $[A2, B2]$, il existe un point d'intersection I si et seulement si l'on a :

$$(yb_2 - ya_2) / (xb_2 - xa_2) = (yb_1 - ya_1) / (xb_1 - xa_1) \text{ et}$$

$$(ya_2(xb_2 - xa_2) - xa_2(yb_2 - ya_2)) / (xb_2 - xa_2) = (ya_1(xb_1 - xa_1) - xa_1(yb_1 - ya_1)) / (xb_1 - xa_1)$$

4. Conclusion

L'emploi de tel ou tel algorithme ne sera pas sans incidence sur votre jeu : les tests doivent être appliqués dans la boucle principale et seront donc effectués à de très nombreuses reprises. Les algorithmes de détection vus dans cet article sont des algorithmes basiques, qui pourront bien entendu être adaptés et optimisés en fonction des éléments que vous manipulerez. Ne négligez pas cette étape de réflexion, car votre jeu pourrait alors être fortement ralenti voire saccadé... ■

Hébergements – Noms de domaines – Emails – VPS – SAAS

Premier hébergeur associatif en France, Web4all est connu et reconnu pour la qualité et les tarifs avantageux de ses offres tout autant que pour la réactivité de son support. Faites confiance à une équipe de passionnés et offrez le meilleur à vos sites Internet !

Lecteurs de Linux-Pratique, une offre vous attend sur www.web4all.fr/LP





L'IMPORTANCE DU GRAPHISME

par Elisa de Castro Guerra

Promenons-nous dans l'univers des styles graphiques pour ensuite plonger dans celui des logiciels libres graphiques...



1. Le game designer

Je ne connais pas un projet web ou un projet de jeu qui ne soit pas à la recherche d'un bon (tant qu'à faire) graphiste. Le design est ce que tout le monde réclame, mais peu sont prêts à rémunérer réellement. Bien souvent, le résultat final est monnayé et le client ignore superbement le temps du graphiste passé à se documenter, à concevoir, imaginer, esquisser. Pourquoi faire ? C'est un artiste ! Il a la chance d'être passionné par son « travail », alors il n'a pas besoin de ça disent quelques clients.

Est-ce que dessiner une commande précise fait du graphiste un artiste ? Est-ce que dessiner, même superbement, l'univers d'un jeu fait du game designer un artiste ? Le designer maîtrise un certain nombre d'outils (logiciels de 2D et/ou de 3D), c'est indéniable et absolument nécessaire. Cependant, cette technique s'apprend. Tout le monde est capable d'apprendre à utiliser ces logiciels. Le développeur du jeu n'a que faire du croquis. Il lui faut le personnage dans un format numérique, sous plusieurs points de vue, réalisant plusieurs actions pour pouvoir animer le tout. La maîtrise des outils est indispensable pour le graphiste, mais pas exclusif. Il est demandé au graphiste de savoir obligatoirement composer, dessiner et d'avoir des connaissances solides en anatomie et en perspectives. Contrairement à ce que l'on pourrait imaginer, concevoir un personnage, même avec des formes légèrement humanoïdes, requiert ce sens des proportions inhérent à l'apprentissage



Fig. 1

de l'anatomie. Pour que le personnage semble plausible, tant dans sa posture que ses mouvements, une connaissance solide sur les principes d'anatomie réelle et des proportions est primordiale.

Un bon graphisme ne suffit pas à faire un bon jeu, mais cela y participe ! À propos des critères d'un bon jeu, je vous invite à lire l'article de Jean-Michel Armand sur la question, présent dans ce numéro. En se focalisant sur le critère unique du graphisme, ce n'est pas aussi simple que l'on pourrait imaginer. La frontière entre un bon graphisme et un mauvais graphisme est ténue. D'une part, nous sommes en face de l'inéluctable différence qui existe entre chacun de nous sur des jugements aussi personnels (comme on dit « les goûts et les couleurs... »). D'autre part, chaque jeu appartient à son époque, a été conçu et pensé sur un support. Juger un jeu créé il y a 20 ans n'est peut-être pas pertinent. D'autant si les supports

d'aujourd'hui ne sont plus adaptés aux jeux d'autrefois. Je pense, par exemple, au pénible affichage de bouillie de pixels renvoyé par l'émulateur sur mon écran haute définition...

2. Les différents styles graphiques dans les jeux

Voici une liste non exhaustive de styles graphiques que l'on trouve actuellement majoritairement dans les jeux.

2.1 Le style cartoon - vectoriel

La plupart des jeux en Flash adoptent ce style. Les jeunes parents doivent connaître le jeu fort sympathique en ligne « Poisson rouge » (Fig. 1), qui utilise cette technologie (<http://www.poissonrouge.com/>).

Le vectoriel comprend des formes complexes stylées par des aplats, des dégradés ou des motifs, redimensionnables sans effet d'escalier. L'avantage du format vectoriel est de s'adapter sur tous les supports, car il peut s'afficher aussi bien sur un petit écran de smartphone ou de tablette que sur un écran haute définition. Le rendu est très sympathique et permet la création d'univers très différents et de personnages ludiques. Pour en savoir plus sur les possibilités du vectoriel dans le cadre du format HTML, je vous renvoie à l'excellent article HTML 5 et SVG de Cédric Gémy dans ce numéro.

En dehors du Web, on retrouve de nombreuses animations et jeux utilisant ce style. Un rendu effet cartoon existe même dans le très bon logiciel graphique libre de 3D, Blender, pour s'approcher de ce style. Concernant la création en 2D de dessin vectoriel, je vous conseille d'utiliser les excellents logiciels libres 2D d'édition vectoriel : Inkscape ou Krita.

2.2 Le style cartoon – manga

Depuis de nombreuses années, les personnages façon manga sont devenus familiers dans nos dessins animés et nos jeux sur écran. Bien souvent, il s'agit d'un style graphique couplé au style vectoriel, mais la prédominance de cette tendance graphique est telle que j'ai estimé important de le mettre en valeur dans une catégorie à part. Ce style de dessin est abordable pour les débutants. Si le dessin de personnages vous intéresse, commencez par des personnages façon manga, avec un logiciel vectoriel ou bitmap et amusez-vous !



Fig. 2 : Une tête façon « manga »

2.3 Le style réaliste avec image de synthèse

Nombreux sont les jeux vidéo arborant ce style. Le plus souvent, cela concerne des jeux à la première personne où on incarne le héros devant accomplir une mission avec une atmosphère oppressante. Il y a une concurrence rude entre les studios utilisant ce style graphique, puisque le jeu le plus réaliste sera le plus prisé. Il n'y a pas le droit à l'erreur. Ce type de style nécessite énormément de ressources de la part du média support. Les mini-jeux sur smartphone exploitent moins ce style. Dans le cadre du HTML 5, le rendu bitmap des images de synthèse sera privilégié.

2.4 Le style réaliste avec photosynthèse

Ce style est présent sur les jeux vidéo plutôt futuristes. Des effets de rendu de particules sont ajoutés aux images de synthèse réaliste. Ces rayons de lumière, lorsqu'ils sont bien maîtrisés, procurent la sensation d'un jeu magnifique. En HTML 5, l'effet équivalent se rencontrera peut-être dans l'intégration d'image bitmap composée avec ce style, judicieusement animée par le CSS 3 et le JavaScript.

2.5 Le style pixel art 2D

Ce style était déjà présent aux pré-mices des jeux sur ordinateur. Au fil du temps, ce style a pu se complexifier. Avez-vous déjà vu le jeu « Pong » ? Quelques pixels clairs sur fond foncé. Un jeu avec du graphisme épuré et pourtant très efficace. Et encore, les jeux « casse-briques » ou « space invader » ne démontrent pas que le graphisme ne fait pas tout ? Aujourd'hui, les jeux comme Zelda, Final Fantasy et même les premières versions de Super Mario Bros sont également en *pixel art*. L'évolution de ce style est également saisissante.

Pour ceux qui souhaiteraient réaliser des dessins dans ce style, favorisez les logiciels graphiques bitmap comme GIMP ou MyPaint, permettant le réglage fin des effets de brosse. La fonctionnalité « Aperçu d'icône » d'Inkscape peut cependant être recherchée, afin de se rendre compte en direct des



Fig. 3 : Exemple de réalisation en pixel art

rendus du dessin en taille réelle. Un post-traitement de l'image peut être ensuite réalisé avec GIMP dans le but de casser l'effet trop lisse du vectoriel.

3. L'atout des logiciels libres graphiques

De nombreux logiciels libres sont utilisés pour développer. Comme le prouvent tous les ans le « Libre Graphics Meeting » et cette année, les Rencontres Mondiales des Logiciels Libres, en consacrant un cycle de conférences et des ateliers sur les logiciels libres graphiques : ceux-ci sont tout à fait capables d'être utilisés par des professionnels du graphisme pour leur travail et ne sont plus seulement réservés aux loisirs. L'association francophone des graphistes libres (l'Afgral, www.afgral.org) le démontre au quotidien, puisque cette association réunit des graphistes professionnels utilisant exclusivement des logiciels libres sous un environnement de travail libre.

Pour clore cet article et laisser des pages aux nombreux autres articles enrichissants de ce magazine, je vous invite à retrouver deux articles en ligne expliquant comment réaliser une image en *pixel art* avec GIMP et un second article montrant comment réaliser une illustration vectorielle avec Inkscape, situés à l'adresse www.yemanjalisa.fr/articles/lp. Concernant la réalisation d'images de synthèse, je vous engage à rechercher par vous-même de la documentation sur le logiciel Blender. ■

Note

Pour ceux qui souhaitent apprendre à utiliser Inkscape, rendez-vous sur fr.flossmanuals.net qui prépare un manuel libre sur ce logiciel. En attendant, consultez en ligne le manuel libre sur l'ePub qui comporte une partie sur le format vectoriel SVG et sur l'utilisation dans le HTML 5 ou préférez la version papier des manuels sur flossmanualsfr.net.



HTML 5 / CSS 3 : pourquoi peut-on maintenant créer des jeux web ?

par Jean-Michel Armand

Les jeux web existent depuis longtemps. Depuis presque aussi longtemps que le Web. Mes premiers souvenirs de jeux web remontent à l'année 2000 avec Fondation, un jeu inspiré par le cycle d'Asimov. Jeu qui d'ailleurs existe toujours. Mais avec HTML 5, la création de jeux vidéo web prend un nouvel essor, nous allons voir pourquoi dans cet article.



Comme je l'ai déjà dit dans l'article parlant de *game design*, il y a deux populations de gens dans le monde des jeux vidéo : les joueurs et les concepteurs. Le

HTML 5 et les possibilités qu'il offre change des choses pour ces deux populations. Nous allons commencer par voir ce qu'il en est au niveau des développeurs et nous terminerons en étudiant ce qu'offre de plus le HTML 5 aux joueurs.

1. Une plateforme technologique complète et standard

Il n'y a de cela pas si longtemps, lorsque l'on voulait faire un jeu vidéo web qui mariait du multijoueur en mode « temps réel », des animations graphiques de qualité et du son, les technologies utilisables étaient plutôt réduites. En fait, il n'y avait pas des technologies mais une, le Flash. Et si on se refusait à utiliser Flash, c'était un peu aller de Charybde en Scylla. En effet, les technologies à la disposition des concepteurs de jeux étaient bien limitées et se bornaient assez souvent à mettre en place des plugins spécifiques à un et unique navigateur, ou alors à se limiter à du très basique, que cela soit en termes de graphisme, d'interaction ou d'effets sonores.

Le HTML 5 vient changer tout cela. Vous verrez dans la suite de ce hors-série que vous allez pouvoir faire des animations graphiques de grande qualité, permettre du véritable multijoueur avec une vraie communication possible en temps réel entre les joueurs. Mais ce n'est pas tout, vous pourrez aussi jouer des sons, des

vrais, et pas vous limiter à quelques petits bruitages cache-misères. Vous aurez enfin accès à une zone de stockage locale et, cerise sur le gâteau, vous pourrez utiliser des graphismes vectoriels en jouant avec des SVG.

De plus, la plateforme HTML 5/CSS 3 est, enfin sera, standardisée. Et donc, les choses devraient à terme fonctionner plus ou moins de la même manière quel que soit le navigateur que vos joueurs utiliseront, tant est que leur navigateur comprend le HTML 5.

2. Une plateforme documentée, avec un écosystème intéressant

J'ai coutume de dire qu'il faut prendre en compte deux critères lorsque l'on veut évaluer un langage ou une technologie. Ce sont ses ressources documentaires et les librairies disponibles. Concernant les ressources, il en existe énormément, que cela soit en libre accès ou payant. Afin de vous donner quelques pistes et de ne pas vous laisser seul avec votre moteur de recherche préféré, en voici quelques-unes :

- Le Planet HTML 5 du W3C [0],
 - Dive Into HTML 5 [1],
 - Les livres de la collection A Book Apart [2],
 - L'excellent livre « HTML 5 » de Rodolphe Rimelé [3],
 - Le flux d'informations HTML5Live [4].

Concernant les librairies utilisables, quand on veut faire un jeu vidéo en HTML 5, il y en a un certain nombre qui vous permettront de justement ne pas refaire la roue. Et cela commence par les moteurs de jeu ou les librairies de gestion du son, de la vidéo ou du SVG. Là encore, pour ne pas vous laisser seul devant votre navigateur, quelques petits liens :

- L'excellent *repository* Github de Bebraw [5], qui liste un grand nombre de moteurs de jeu JS pour HTML 5, mais répertorie aussi un grand ensemble de ressources sur le développement de jeux vidéo HTML 5,
 - RPG JS [6], un moteur JS/HTML5 pour faire un RPG et en licence GPL,
 - LimeJS [7], un framework pour jeux vidéo HTML 5, sous licence Apache (avec un *repository* Github),
 - MelonJS [8], un framework pour jeux vidéo réputé pour être léger et compatible avec un éditeur de niveaux en mode Tiles.

3. Une plateforme vivante et en pleine ébullition

Chaque semaine, il fleurit un nouveau blog sur la toile qui parle de HTML 5, de CSS 3 ou de SVG. Chaque jour, des développeurs posent des questions et d'autres leur proposent des réponses. Il y a pléthora de tutoriels pour apprendre jusqu'à plus soif, en fait il y en a tellement que cela pourrait devenir un problème, tellement on pourrait se retrouver à apprendre sans cesse et du coup, ne plus travailler

sur son projet. Donc oui, la communauté des développeurs HTML 5 est très active. Y compris dans le domaine du jeu vidéo. Pour ne parler que de quelques expériences qui donnent une certaine idée de ce qui est faisable :

- Arcade fire [9], une expérience artistique qui mêle vidéo, musique, image de Google Street View. Très intéressant à regarder ;
- D.E.M.O [10], une démonstration toute récente (début août 2012) de ce que l'on peut obtenir en termes de FPS 3D en HTML 5 et dans son navigateur. D.E.M.O a été créé par Playcanvas. Elle utilise WebGL, Web Audio API, les WebSockets et Gamepad API. Même si les graphismes ont un certain âge, il faut bien avouer que la démonstration est assez bluffante ;
- BrowserQuest [11]. Un Zelda-like, fait par la fondation Mozilla, de la grande époque dans votre browser. Parcourez le monde, tuez des monstres, rencontrez les autres joueurs, tout cela dans un style très *oldies*. Le tout est fait en HTML 5 et NodeJS. Là aussi, très impressionnant.

4. Une plateforme qui peut permettre plus

L'arrivée du HTML 5 ouvre des possibilités qu'il n'y avait pas ou difficilement précédemment. L'utilisation de la géolocalisation est un excellent exemple de cela. Il n'aura jamais été aussi facile d'intégrer dans un jeu vidéo des données de géolocalisation pour améliorer l'immersion. Mais plus encore, le HTML 5 offre une grande liberté pour tout ce qui touche au multi-plateforme. Il est en effet possible de concevoir un jeu qui pourra débuter dans le navigateur d'un de vos joueurs, continuer sur sa tablette, reprendre sur sa box TV et se poursuivre sur son téléphone lorsqu'il prendra le bus. Et en réfléchissant un peu, on peut utiliser cette possibilité de multi-plateforme pour offrir aux joueurs une expérience inédite.

Tentons de prendre un exemple. Imaginons un jeu d'exploration et conquête. Construire une base, des infrastructures de traitement de ressources et de construction d'armées, un RTS en clair. Jusque-là, rien d'innovant, ou qui tire parti des différences entre les devices utilisables pour jouer. J'y arrive. Pour construire une nouvelle base, on peut imaginer qu'il faudrait amener une unité

spéciale, une base mobile d'exploration sur les lieux de la future base et la déployer. On pourrait se dire qu'il serait intéressant au niveau game design de ne permettre cette action qu'à partir d'un téléphone portable. On pourrait alors utiliser la géolocalisation du téléphone pour positionner la base mobile d'exploration. Il faudrait donc sortir de chez soi, utiliser l'interface mobile du jeu et un peu de réalité augmentée et voilà !

Je pourrais développer le principe plus loin, réfléchir à ce que l'on pourrait ajouter dans le cas où un joueur en mode smartphone en croise un autre dans le même mode ; pourquoi pas alors un mode FPS ou alors un combat en format jeu de cartes ? Mais cela déborderait du cadre de cet article, qui n'est pas comment bien concevoir un jeu multi-plateforme, mais pourquoi le HTML 5 est une vraie solution de développement de jeux vidéo.

Réalité augmentée

La réalité augmentée désigne le fait d'ajouter des informations 2D ou 3D sur la perception que nous avons de la réalité et cela, en temps réel. La réalité augmentée peut être utilisée dans de multiples domaines, comme par exemple ajouter des informations touristiques visibles à travers un smartphone et qui se déposeraient au-dessus des bâtiments intéressants lorsqu'on braquerait la caméra de son téléphone devant soi. On peut aussi l'utiliser dans le domaine des jeux vidéo. On pourrait par exemple imaginer un jeu de survie à jouer avec son téléphone et qui ferait apparaître des aliens dans les embrasures de portes ou sous les plaques d'égouts.

5. Conclusion

Nous venons donc de voir en quoi le HTML 5 est aujourd'hui un véritable choix de développement de jeu vidéo, pour ne pas dire un excellent choix. En introduction, je vous avais promis d'étudier les choses du point de vue du concepteur de jeux vidéo, mais aussi de celui du joueur. M'étant très largement étendu en ce qui concerne le développeur, je vais profiter de cette conclusion pour parler du joueur. Je ne vais pas faire une partie spécifique pour le joueur, parce qu'au final, cela serait en grande partie une redite de la partie développeur.

Concernant le joueur de jeu web, il faut bien le dire, jusqu'à présent, jouer à des jeux web a bien souvent été une plaie. C'était souvent presque aussi compliqué que de trouver la bonne version de Wine, celle qui nous permettrait de jouer à un de nos jeux Windows. Le HTML 5, avec sa standardisation future, devrait arranger grandement cela. De même, on pourra espérer profiter d'une augmentation de la qualité dans tous les domaines ; les qualités graphiques et sonores étant jusqu'à présent plusieurs crans en-dessous de ce que l'on pouvait trouver dans les jeux vidéo classiques. Entendons-nous bien à ce sujet, je ne suis pas pour une prédominance des graphismes dans les jeux vidéo, bien au contraire, je suis en fait un grand fan des jeux textuels. Mais lorsque un jeu propose des graphismes à ses joueurs, autant qu'il propose quelque chose de correct. Enfin, et surtout, l'utilisation du HTML 5 va permettre d'offrir aux joueurs, si les concepteurs sont à la hauteur, une palette de choix comme ils n'en ont pas souvent eu.

Je disais il y a trois lignes que j'étais un fan des jeux textuels. Apparemment, je ne suis pas le seul, un site (Playfic [12]) regroupant de tels jeux vient d'ailleurs d'être lancé. Des revivals de vieux styles de jeux, des jeux *oldschool*, de l'intégration de la géolocalisation, du multi-plateforme, etc., voilà ce que peuvent nous promettre les jeux en HTML 5 ! Espérons que toutes ces promesses vont se réaliser... ■

Références

- [0] Planet HTML 5 : <http://www.w3.org/html/planet/>
- [1] Dive into HTML 5 : <http://diveintohtml5.info/>
- [2] A Book Apart : <http://www.abookapart.com/>
- [3] HTML5, chez Eyrolles : <http://www.eyrolles.com/Informatique/Livre/html5-9782212129823>
- [4] Html5live : <http://html5live.fr/>
- [5] <https://github.com/bebrav/jswiki/wiki/Game-Engines>
- [6] RPGJS : <http://rpgjs.com/>
- [7] LimeJS : <http://www.limejs.com/>
- [8] MelonJS : <http://www.melonjs.org/>
- [9] Arcade fire : <http://thewildernessdowntown.com/>
- [10] D.E.M.O : <http://apps.playcanvas.com/playcanvas/scifi/latest>
- [11] BrowserQuest : <http://browserquest.mozilla.org/>
- [12] Playfic : <http://playfic.com/>



GRAPHISMES : utilisation des canvas

par Tristan Colombo

Une zone spéciale permettant d'accueillir les graphismes est apparue en HTML 5 : la balise `<canvas>`. C'est grâce à cette balise que l'on a pu voir apparaître de nombreux jeux et que l'on peut désormais dessiner facilement sur une page web.



L'affichage des écrans de n'importe quel jeu passera par au moins une balise `<canvas>`. Mais au-delà de l'affichage de simples images, certains jeux permettent de dessiner. Sur les plateformes mobiles, ces jeux se sont rapidement répandus et on peut y trouver par exemple « Cut the Rope » (disponible en HTML 5 [1]), ou des jeux vous demandant de dessiner un parcours pour déplacer un objet vers un point final (« Crayon Physics », etc.).

Cet article permettra de découvrir ou redécouvrir la balise `<canvas>`, de voir comment y dessiner, y écrire ou encore comment y afficher des images. Pour vraiment élargir le champ d'utilisation des `canvas` aux jeux, nous étudierons ensuite comment détecter la position de la souris et, pour conclure, nous verrons comment réaliser un effet de *scrolling* horizontal ou vertical (ou même en diagonale si vous en avez envie).

1. La balise `<canvas>` de base

Dans son utilisation la plus simple possible, la balise `<canvas>` ne prendra qu'un seul attribut `id` lui donnant un identifiant pour être manipulée par la suite en JavaScript. Le code suivant introduit ainsi une telle balise :

```
01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test sur les canvass</title>
06:   </head>
07:   <body>
08:     <canvas id="drawing"></canvas>
09:   </body>
10: </html>
```

Malheureusement, ce code ne produira aucun affichage : par défaut un `canvas` a une taille de 300 px de long pour 150 px de haut et sa bordure n'est pas affichée. Pour modifier la taille du `canvas`, il vous faudra utiliser les attributs `width` et `height` :

```
<canvas id="drawing" width="768px"
height="400px"></canvas>
```

Pour afficher le cadre de la zone de dessin, une règle CSS suffira :

```
#drawing
{
  border: 1px solid #000;
}
```

En HTML pur, nous ne pourrons pas aller plus loin. Pour vraiment utiliser cet objet, il va falloir passer par l'API de dessin en JavaScript.

2. Dessiner et écrire

Avant de pouvoir manipuler notre `canvas`, il va falloir réaliser deux opérations fondamentales :

- récupérer un lien vers le nœud de notre `canvas` :

```
var drawing = document.
getElementById("drawing");
```

- créer une variable de « contexte » qui va faire le lien avec l'API de dessin en deux dimensions :

```
var ctx_drawing = drawing.
getContext("2d");
```

Une fois ces étapes effectuées, nous travaillerons sur la variable de contexte pour réaliser nos dessins.

L'origine des coordonnées permettant de se positionner dans la zone de dessin se trouve en haut à gauche comme le montre la figure 1.

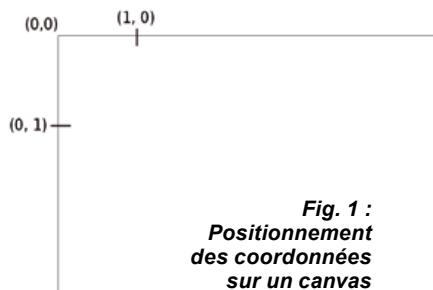


Fig. 1 :
Positionnement
des coordonnées
sur un canvas

2.1 Dessiner un rectangle

On peut dessiner soit un rectangle plein, soit seulement la bordure à l'aide de deux méthodes acceptant les mêmes paramètres : les coordonnées (x, y) du coin supérieur gauche du rectangle et la taille du rectangle (longueur et hauteur). Le prototype de ces méthodes est le suivant :

- Rectangle plein : `fillRect(x, y, width, height)` ;
- Rectangle (bordure) : `strokeRect(x, y, width, height)`.

Pour afficher un rectangle plein de taille 100 px x 150 px aux coordonnées (10, 10), il faudra donc écrire le code suivant, que nous placerons dans un fichier `draw_canvas.js` :

```

01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:     ctx_drawing.fillRect(10, 10, 100, 100);
06: }
07: );

```

Ce code fait appel à jQuery pour indiquer que les commandes ne devront être exécutées qu'une fois tous les éléments de la page chargés. En effet, si l'élément <canvas> n'a pas encore été créé, nous ne pourrons rien y afficher. Le code HTML de chargement de jQuery devra être inséré dans le code HTML présenté précédemment :

```

05_1: <script src="script/jquery-1.7.2.min.js"></script>
05_2: <script src="script/draw_canvas.js"></script>

```

Le rectangle obtenu est noir... Il s'agit de la couleur de remplissage par défaut. Comme dans les logiciels de dessin, les **canvas** définissent une couleur de remplissage, mais également une couleur de dessin qui est différente. Pour modifier ces valeurs, il faudra indiquer un code couleur au format RGB dans des attributs de l'élément contexte :

- Changer la couleur de dessin : `strokeStyle = "#rgb"`
- Changer la couleur de remplissage : `fillStyle = "#rgb"`

Si nous voulons dessiner un rectangle gris et sa bordure en rouge comme sur la figure 2, il nous faudra exécuter le code suivant :

```

01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:     ctx_drawing.strokeStyle = "#f00";
06:     ctx_drawing.fillStyle   = "#9e9e9e";
07:
08:     ctx_drawing.fillRect(10, 10, 100, 100);
09:     ctx_drawing.strokeRect(10, 10, 100, 100);
10: }
11: );

```

Les lignes 5 et 6 permettent de définir les couleurs de dessin et de remplissage, puis en ligne 8, nous dessinons un rectangle plein et en ligne 9, aux mêmes coordonnées, nous dessinons le bord du rectangle en rouge.

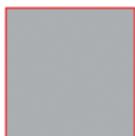


Fig. 2 : Dessin d'un rectangle plein et d'une bordure de rectangle dans un canvas

Les rectangles permettent également de définir des zones du **canvas** à effacer. Il faut pour cela appeler la méthode `clearRect()`. En passant en paramètres les coordonnées de l'origine (0, 0) et la taille du canvas, nous effaçons complètement la zone de dessin :

```
ctx_drawing.clearRect(0, 0, 768, 400);
```

2.2 Dessiner un cercle

Les cercles ou les arcs de cercle sont définis à l'aide d'une même méthode `arc()`. Cette méthode prend en paramètres les coordonnées du centre du cercle, son rayon, l'angle de départ et l'angle d'arrivée de l'arc de cercle (en radians) et enfin, un paramètre booléen permettant d'indiquer si vous souhaitez tracer votre arc dans le sens des aiguilles d'une montre (`false`) ou dans le sens inverse (`true`).

Les angles étant exprimés en radians, il va falloir utiliser la constante pi (`Math.PI`). Comme nous parlons ici d'arcs, pour dessiner un cercle il va falloir indiquer un angle de 2 pi :

```
arc(100, 100, 50, 0, 2*Math.PI, false) ;
```

Dans l'exemple suivant, nous dessinons un disque (cercle plein), un cercle et un arc de cercle. Le résultat de ce code est présenté en figure 3.

```

01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:     ctx_drawing.lineWidth = 4;
06:
07:     ctx_drawing.beginPath();
08:     ctx_drawing.arc(100, 100, 50, 0, 2*Math.PI, false);
09:     ctx_drawing.fill();
10:
11:    ctx_drawing.beginPath();
12:    ctx_drawing.strokeStyle = "#f00";
13:    ctx_drawing.arc(125, 125, 25, 0, 2*Math.PI, false);
14:    ctx_drawing.stroke();
15:
16:    ctx_drawing.beginPath();
17:    ctx_drawing.strokeStyle = "#00f";
18:    ctx_drawing.arc(100, 100, 54, Math.PI/2, true);
19:    ctx_drawing.stroke();
20: }
21: );

```

Dans les lignes 7 à 9, nous définissons un disque noir centré en (100, 100) et de rayon 50 px. La différenciation disque/cercle se fait en ligne 9, en utilisant la méthode `fill()` pour remplir le cercle. Dans les lignes 11 à 14, nous définissons un cercle rouge (ligne 12) de centre (125, 125) et de rayon 25 px. Enfin, dans les lignes 16 à 19, nous créons un arc de cercle bleu (ligne 17) d'angle (0, pi/2) en sens inverse des aiguilles d'une montre.

Dans les lignes 7, 11 et 16 nous avons utilisé la méthode `beginPath()` permettant de définir un chemin. Elle permet de bien séparer les différents tracés. Son utilisation sera détaillée dans la partie suivante.



Fig. 3 : Tracé d'arcs de cercles dans un canvas

2.3 Dessiner des lignes

Le dessin de lignes se fait un peu à la mode logo : déplacer le crayon jusqu'à un point et dessiner jusqu'à un autre point. Positionner le crayon se fait à l'aide de la méthode `moveTo()`

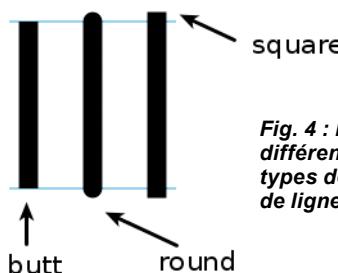


Fig. 4 : Les différents types de fin de ligne

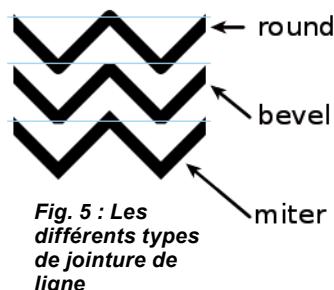


Fig. 5 : Les différents types de jointure de ligne

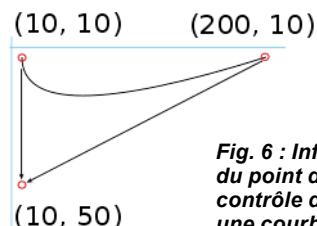


Fig. 6 : Influence du point de contrôle dans une courbe de Bézier quadratique

et dessiner jusqu'à un point se fait à l'aide de la méthode `lineTo()`. L'affichage effectif dans la zone de dessin se fera par un appel à `stroke()` (ou à `fill()` si vous souhaitez que la forme définie soit remplie).

En général, des lignes définissent un « chemin » et on commence toujours une suite de lignes définissant une forme par la méthode `beginPath()`. L'utilisation de cette méthode permet éventuellement d'utiliser la méthode `closePath()` en fin de tracé : une ligne sera alors tracée du point courant jusqu'au point de départ pour fermer la figure.

Pour appliquer ces méthodes, dessinons par exemple un « Z » :

```
01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:
06:     ctx_drawing.beginPath();
07:     ctx_drawing.moveTo(10, 10);
08:     ctx_drawing.lineTo(100, 10);
09:     ctx_drawing.lineTo(10, 100);
10:     ctx_drawing.lineTo(100, 100);
11:     ctx_drawing.stroke();
12: }
13: );
```

En ligne 6, nous commençons notre tracé et en ligne 7 nous nous déplaçons au point de coordonnées (10, 10), puis dans les lignes 9 à 10, nous traçons une ligne jusqu'au point (100, 10), puis encore une vers le point (10, 100) et enfin, une dernière vers le point (100, 100). Si nous nous limitons à ces lignes, il n'y aura aucun affichage... Il ne faut pas oublier la ligne 11 qui affiche notre tracé à l'écran.

L'épaisseur du trait peut être modifiée par l'attribut `lineWidth` et le style de fin de ligne peut être défini par l'attribut `lineCap` acceptant trois valeurs (`butt`, `round` ou `square`) illustrées en figure 4 :

- `butt` termine la ligne immédiatement à la position spécifiée ;
- `round` ajoute un « chapeau » arrondi ayant pour rayon la moitié de l'épaisseur de la ligne ;
- `square` ajoute un « chapeau » rectangulaire de la même longueur que l'épaisseur de la ligne et ayant pour hauteur la moitié de l'épaisseur de la ligne.

Lorsque deux lignes se rejoignent, on peut également modifier le style de jointure à l'aide de l'attribut `lineJoin` (valeurs `round`, `bevel` ou `miter`). La figure 5 illustre les cas possibles :

- `round` arrondit les angles des lignes avec un rayon égal à l'épaisseur de la ligne ;
- `bevel` ne fait rien... ;
- `miter` prolonge le tracé des lignes de manière à ce qu'elles se rejoignent parfaitement et ne forment qu'un seul angle.

2.4 Dessiner des formes complexes

Pour les formes complexes, l'API de dessin met à notre disposition deux méthodes permettant de dessiner des courbes de Bézier :

- `quadraticCurveTo()` : courbe de Bézier quadratique ayant un point de départ, un point d'arrivée et un seul point de contrôle ;
- `bezierCurveTo()` : courbe de Bézier ayant un point de départ, un point d'arrivée et deux points de contrôle.

La manipulation de ces courbes n'est pas très aisée, car contrairement aux logiciels de dessin, nous n'avons aucune vérification visuelle sur les points de contrôle. Voici un exemple très simple représentant l'influence du point de contrôle : nous allons dessiner une courbe partant du point de coordonnées (10, 10) jusqu'à (200, 10), avec pour point de contrôle le point (10, 70). Ce point va en quelque sorte attirer la courbe vers lui, comme l'illustre la figure 6. Le code permettant de réaliser ce dessin est le suivant :

```
01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:
06:     ctx_drawing.beginPath();
07:     ctx_drawing.moveTo(10, 10);
08:     ctx_drawing.quadraticCurveTo(10, 70, 200, 10);
09:     ctx_drawing.stroke();
10: }
11: );
```

Le point de départ est donné en ligne 7 par le positionnement du « stylo » et en ligne 8, nous indiquons le point de contrôle et le point d'arrivée.

2.5 Écrire

Comme pour tout texte affiché à l'écran, l'insertion de texte dans une zone de dessin va demander de sélectionner une police et éventuellement un type d'alignement :

- **font** : choix de la police (exemple "20pt Arial Bold");
- **textAlign** : choix de l'alignement (**left**, **right**, **center**, **start** ou **end**). Le texte sera inséré au point courant (spécifié par la méthode d'affichage du texte) avec des modifications appliquées par l'alignement :
 - **left** ou **start** : le début du texte se trouve au point courant ;
 - **right** ou **end** : la fin du texte se trouve au point courant ;
 - **center** : le texte est centré sur la position du point courant.

L'affichage du texte se fait ensuite par la méthode **fillText()** pour du texte « plein », ou par **strokeText()**. Voici un exemple affichant « Linux Pratique » :

```
01: $(document).ready(function()
02: {
03:   var drawing = document.getElementById("drawing");
04:   var ctx_drawing = drawing.getContext("2d");
05:
06:   ctx_drawing.font = '50pt Arial Bold';
07:   ctx_drawing.textAlign = 'left';
08:   ctx_drawing.strokeText('Linux Pratique', 100, 100);
09: }
10: );
```

Lorsque le texte n'est pas rempli (ligne 8 appel de **strokeText()**) et que la police a une grande taille (ligne 6), on obtient le contour des lettres (Fig. 7).

Des effets d'ombrage peuvent être appliqués sur le texte. Il faut alors choisir la taille de l'ombre sur x (**shadowOffsetX**) et sur y (**shadowOffsetY**), régler l'effet de dispersion avec l'attribut **shadowBlur** et enfin, choisir la couleur de l'ombre avec **shadowColor**.

L'exemple suivant reprend l'affichage du texte « Linux Pratique » en ajoutant une ombre (lignes en rouge) :

```
01: $(document).ready(function()
02: {
03:   var drawing = document.getElementById("drawing");
04:   var ctx_drawing = drawing.getContext("2d");
05:
06:   ctx_drawing.font = '50pt Arial Bold';
07:   ctx_drawing.textAlign = 'left';
08:   ctx_drawing.shadowOffsetX = 10;
09:   ctx_drawing.shadowOffsetY = 10;
10:   ctx_drawing.shadowBlur = 5;
11:   ctx_drawing.shadowColor = '#22e';
12:   ctx_drawing.strokeText('Linux Pratique', 100, 100);
13: }
14: );
```

L'ajout de cette ombre produit le résultat de la figure 8.



Fig. 7 : Affichage de texte dans un canvas

Linux Pratique

Fig. 8 : Affichage de texte avec ombre dans un canvas

3. Afficher une image

L'affichage d'une image sera un point essentiel pour la plupart des jeux. Pour charger une image, il faudra créer un objet **Image** et spécifier le nom du fichier à télécharger grâce à l'attribut **src** :

```
var monImage = new Image();
monImage.src = 'fichier_image.png';
```

Pour afficher cette image dans notre **canvas**, il faudra utiliser la méthode **drawImage()** sur le contexte. Cette méthode peut s'utiliser en omettant certains de ses paramètres :

- **drawImage(image, x, y)** : affiche l'angle supérieur gauche de l'image en position (**x, y**) ;
- **drawImage(image, x, y, width, height)** : affiche l'angle supérieur gauche de l'image en position (**x, y**) et redimensionne l'image avec une longueur de **width** et une hauteur de **height** ;
- **drawImage(image, src_x, src_y, src_width, src_height, x, y, width, height)** : affiche une portion de l'image (l'angle supérieur gauche de la portion d'image est en position (**src_x, src_y**) et sa taille est **src_width x src_height**) en position (**x, y**) et redimensionne l'image avec une longueur de **width** et une hauteur de **height** .

Prenons le logo de *Linux Pratique* de la figure 9 et affichons seulement le petit pingouin se situant à gauche dans le cadre.

```
01: $(document).ready(function()
02: {
03:   var drawing = document.getElementById("drawing");
04:   var ctx_drawing = drawing.getContext("2d");
05:
06:   var img = new Image();
07:   img.src = 'logo_lp.jpg';
08:
09:   ctx_drawing.drawImage(img, 1, 8, 80, 88, 100, 100, 80, 88);
10: }
11: );
```

L'objet **Image** est créé en ligne 6, puis chargé en ligne 7. En ligne 9, nous affichons une portion de cette image (80 px de long pour 88 px de haut, avec pour point supérieur gauche (1, 8)) à l'échelle 1:1 en position (100, 100).

Il est possible que l'image n'apparaisse pas et qu'il faille recharger la page... Tout simplement car au moment d'insérer l'image dans le **canvas**, celle-ci n'a pas été totalement téléchargée. Les objets **Image** admettent un attribut permettant de gérer ce problème : **onload** qui accepte une fonction en valeur. La ligne 9 peut alors être modifiée en :

```

09: img.onload = function()
10: {
11:     ctx_drawing.drawImage(img, 1, 8, 80, 88, 100, 100, 80, 88);
12: };

```

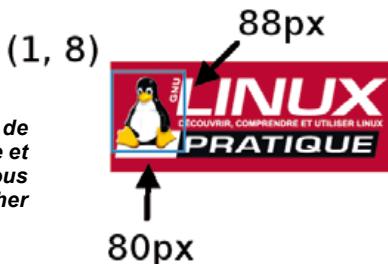


Fig. 9 : Logo de Linux Pratique et zone que nous souhaitons afficher

L'API de dessin permet également de récupérer un tableau de pixels correspondant à une zone du **canvas**. Les informations sur une portion de la zone de dessin seront récupérées par la méthode **getImageData()**, qui prend en paramètres les coordonnées (x, y) du point correspondant à l'angle supérieur gauche de la zone, suivies de la longueur et de la hauteur de la zone.

Chaque pixel comporte quatre valeurs informatives : les trois composantes RGB et la valeur de transparence alpha. Ces valeurs sont des entiers compris entre 0 et 255. Pour la composante alpha, 255 signifie une opacité totale. Un pixel qui n'aura pas été assigné aura pour valeur (0, 0, 0, 0). Si vous modifiez les valeurs de certains pixels, vous pourrez transmettre ces modifications pour affichage grâce à la méthode **putImageData()**.

Reprendons l'exemple précédent : si nous récupérons la zone en (100, 100) ayant pour taille 80 px x 88 px, nous aurons les informations correspondant aux pixels de notre image et nous pourrons alors rendre celle-ci transparente :

```

01: $(document).ready(function()
02: {
03:     var drawing = document.getElementById("drawing");
04:     var ctx_drawing = drawing.getContext("2d");
05:
06:     var img = new Image();
07:     img.src = 'logo_lp.jpg';
08:
09:
10:    img.onload = function()
11:    {
12:        ctx_drawing.drawImage(img, 1, 8, 80, 88, 100, 100, 80, 88);
13:
14:        var imgd = ctx_drawing.getImageData(100, 100, 80, 88);
15:        var pix = imgd.data;
16:        for (var i=3; i<pix.length; i+=4)
17:        {
18:            pix[i] = 150;
19:        }
20:
21:        ctx_drawing.putImageData(imgd, 100, 100);
22:    };
23: }
24: );

```

Les lignes 14 à 21 permettent de récupérer les informations, de les modifier et de les afficher. Notez bien que ces lignes

sont placées après l'affichage de l'image en (100, 100), car sinon, les informations seraient récupérées avant l'affichage et correspondraient donc à des pixels non alloués (0, 0, 0, 0). La ligne 14 correspond à la lecture effective des informations de la zone de dessin. En ligne 15, nous créons une variable intermédiaire qui fait directement le lien sur le tableau de pixels stocké dans l'attribut **data** de la variable **imgd**. La boucle des lignes 16 à 19 permet de parcourir tout le tableau et affecte la valeur **150** à tous les indices correspondant à la composante alpha des pixels (3, 3+4, 3+4+4, etc.).

Il faut savoir que le tableau **pix** a une taille correspondant à la longueur de la zone capturée x la hauteur de la zone x 4. Enfin, la ligne 21 permet de porter les modifications à l'écran.

L'analyse du tableau de pixels d'une zone permettra par exemple de détecter des collisions entre objets...

4. Détection de la souris

La position de la souris peut être récupérée en activant l'écoute des événements souris. Le code suivant est à insérer dans la fonction anonyme de la méthode **ready()**, de manière à être exécuté au chargement de la page :

```

01: $('#drawing').mousedown(function(event)
02: {
03:     var mouseX;
04:     var mouseY;
05:
06:     if ((event.x != undefined) && (event.y != undefined))
07:     {
08:         mouseX = event.x;
09:         mouseY = event.y;
10:     }
11:     else
12:     {
13:         mouseX = event.clientX + document.body.scrollLeft +
14:             document.documentElement.scrollLeft;
15:         mouseY = event.clientY + document.body.scrollTop +
16:             document.documentElement.scrollTop;
17:     }
18:     mouseX -= this.offsetLeft;
19:     mouseY -= this.offsetTop;
20:
21:     console.log(mouseX + ', ' + mouseY);
22: }
23: );

```

La ligne 1 active l'écoute de l'événement clic de souris dans la zone du **canvas** identifié par **drawing**. Nous stockerons les coordonnées de la souris dans des variables **mouseX** et **mouseY** (lignes 3 et 4). Les lignes 6 à 10 permettent de récupérer la position du pointeur de la souris pour les navigateurs autres que Firefox, et la méthode spéciale pour Firefox se trouve dans les lignes 12 à 17.

Pour que ces coordonnées soient relatives au **canvas**, il faut les recentrer dans le **canvas**. C'est le travail des lignes 18 et 19. Pour finir, nous affichons un message dans la console de Firebug avec les coordonnées du pointeur dans le **canvas** (ligne 21).

5. Réaliser un effet de scrolling horizontal ou vertical

La réalisation d'un scrolling horizontal ou vertical est très simple : nous allons afficher une image dans un `canvas` trop petit qui se comportera alors comme une fenêtre permettant de voir une partie de l'image. Ensuite, en effectuant une translation progressive de ce `canvas`, nous créerons un effet de déplacement.

Il y a deux façons de réaliser cette translation : utiliser la méthode `translate()` sur le `canvas` ou alors ré-afficher l'image de fond en la décalant. La méthode `translate()` ne déplace pas le `canvas` lui-même, mais la zone de dessin qui y est incorporée. Pour qu'un déplacement soit apparent, il faut afficher une nouvelle image après appel à cette méthode. Quitte à afficher une nouvelle fois l'image, je trouve plus simple de lui indiquer directement ses nouvelles coordonnées relatives et c'est cet exemple que nous allons étudier. Il faut tout de même savoir que l'utilisation de `translate()` fournira le même résultat.

```

01: function scrolling(ctx_drawing, img)
02: {
03:   if (scrolling.pos.x == -832)
04:   {
05:     scrolling.dir = 'left';
06:   }
07:   if (scrolling.pos.x == 0)
08:   {
09:     scrolling.dir = 'right';
10:   }
11:   if (scrolling.dir == 'right')
12:   {
13:     scrolling.pos.x--;
14:   }
15:   else
16:   {
17:     scrolling.pos.x++;
18:   }
19:   ctx_drawing.clearRect(0, 0, 768, 400);
20:   ctx_drawing.drawImage(img, scrolling.pos.x, 0);
21: };
22:
23: $(document).ready(function()
24: {
25:   var drawing = document.getElementById("drawing");
26:   var ctx_drawing = drawing.getContext("2d");
27:
28:   var img = new Image();
29:   img.src = 'stars.jpg';
30:
31:
32:   img.onload = function()
33:   {
34:     ctx_drawing.drawImage(img, 0, 0);
35:   };
36:
37:   scrolling.pos = { 'x': 0, 'y': 0 };
38:   scrolling.dir = 'right';
39:   setInterval(function() { scrolling(ctx_drawing, img); }, 1000/30);
40: }
41: );

```

Le code ci-dessus correspond au code vu précédemment pour afficher une image (lignes 23 à 35), auquel nous avons ajouté le déplacement de l'image. Ce déplacement se fait grâce à la fonction `scrolling()` des lignes 1 à 21. La fonction

`scrolling()` étant un objet, nous lui ajoutons deux attributs `pos` et `dir` indiquant la position de l'image et la direction du scrolling. En ligne 39, nous appelons à intervalles réguliers la fonction `scrolling()`.

L'utilisation de la fonction `setInterval()` sera vue plus en détail dans l'article suivant sur le déplacement des *sprites*. L'image que nous affichons ici a une taille de 1600 px x 800 px et notre `canvas` a une taille de 768 px x 400 px : le déplacement total vers la droite de notre fenêtre sera alors de $1600 - 768 = 832$ px. Le déplacement relatif de l'image est défini par la valeur de la variable `scrolling.pos.x` et lorsque l'on arrive au bout de l'image (décalage de 832 px), on repart en scrolling vers la gauche et ainsi de suite. Tout ceci est géré par les lignes 3 à 18, la ligne 19 permettant d'effacer la zone de dessin et la ligne 20 permettant de ré-afficher l'image.

Nous avons réalisé ici un scrolling horizontal « simple » dans les deux sens. En jouant uniquement sur les coordonnées en abscisse, vous obtiendrez un scrolling vertical et en faisant évoluer les deux coordonnées, vous pourrez créer un effet de scrolling en diagonale, en zig-zag, etc.

6. Conclusion

Nous avons réalisé un tour d'horizon de différentes techniques de dessin et d'animation de graphismes en deux dimensions. Pour la 3D, ça n'est pas pour tout de suite... Vous avez sans doute vu de nombreuses démonstrations d'affichage 3D en HTML 5 sur le Web : la plupart sont conçues par calcul à partir d'un contexte en deux dimensions et les autres utilisent une version bêta de l'implémentation 3D avec WebGL... Mais malheureusement, tous les navigateurs ne supportent pas WebGL et cette fonctionnalité est encore expérimentale. Nous reviendrons certainement sur cet aspect du dessin en HTML 5 dans un prochain numéro. ■

Référence

[1] Jeu « Cut the Rope » en HTML 5 :
<http://www.cuttherope.ie/>



ANIMER UN SPRITE

par Tristan Colombo

Le sprite est à la base de l'animation dans un jeu vidéo. Il s'agit d'une image contenant tous les mouvements d'un personnage que l'on affichera successivement pour produire une impression de mouvement. En HTML 5, les « canvas » permettent de réaliser cela simplement.



Le déplacement de personnages (joueurs ou ennemis) est bien souvent la base d'un jeu vidéo. Dans cet article, nous allons analyser la manière d'animer des sprites en HTML 5 grâce à la balise `<canvas>` étudiée dans un article précédent du présent hors-série. La création de sprites étant une tâche fastidieuse dévolue généralement aux graphistes, je n'ai réalisé aucun des sprites du présent article. J'ai simplement utilisé des sprites issus de la librairie SpritesLib distribuée sous licence CPL (*Common Public Licence*) : <http://www.widgetworx.com/widgetworx/portfolio/spritelib.html>. Ces sprites ont été légèrement retravaillés sous GIMP de manière à obtenir un fond transparent. Ceci permet de se concentrer sur le code et éventuellement de faire appel à un graphiste pour reprendre les graphismes une fois le développement fini.

Nous commencerons par définir ce qu'est l'animation d'un sprite avant de le déplacer réellement dans un `canvas`. Pour conclure, nous verrons comment faire évoluer un sprite dans un décor en plusieurs plans.

1. Le principe de l'animation par les sprites

Les sprites sont de petites images représentant les différentes étapes de l'animation d'un personnage. La figure 1 montre une page de sprites décomposant les mouvements de 16 personnages. Chaque sprite a une taille de 55 x 55 pixels. Pour pouvoir afficher une image particulière, il suffit donc de connaître les coordonnées du point supérieur gauche de l'image. Dans l'exemple de la figure 1, on voit que l'image du chien ouvrant de grands yeux se trouve aux coordonnées $x = 424$ et $y = 4$. Connaissant ce point et la taille du sprite, nous pouvons afficher l'image.

Vous remarquerez sur cette page que les mouvements des personnages ne sont composés que de deux images. Toujours en prenant le chien, le déplacement vers la gauche se décompose en deux images : la première et la seconde du rectangle rouge.

Fig. 1 :
Page de
sprites



de la figure 1. Nous obtiendrons donc une animation très simple du mouvement (mais tout de même acceptable) en alternant l'affichage de ces images suivant un rythme plus ou moins rapide, noté FPS pour *Frames Per Second* (nombre d'images affichées par seconde). Au plus le mouvement sera décomposé en un grand nombre d'images, au plus le déplacement paraîtra naturel. La figure 2 montre ainsi le déplacement d'un félin décomposé en six images (au lieu de deux !).

Ces deux pages de sprites sont issues de la librairie SpritesLib citée précédemment. Ces images ne sont donc pas destinées essentiellement à une utilisation pour du développement web. Les déplacements des personnages ne sont détaillés que dans un seul sens : vers la gauche pour le petit chien de la figure 1 et vers la droite pour le tigre de la figure 2. Pour obtenir les images du déplacement dans le sens inverse, il faut donc retourner le sprite (effectuer une symétrie axiale ou *flip* en anglais). Cette opération pourra être réalisée en CSS 3 avec la règle `transform` :

```
transform : matrix(-1, 0, 0, 1, 0, 0);
```

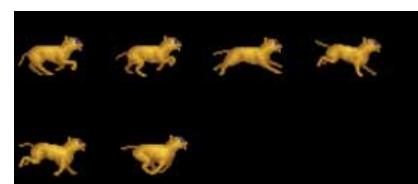


Fig. 2 : Page de sprites illustrant le déplacement d'un tigre à dents de sabre avec une décomposition fine du mouvement



Fig. 3 : Page de sprites contenant des blocs de décor

Une autre solution consistera à ajouter les sprites de déplacement en sens inverse sur la page de sprites. C'est l'éternel dilemme temps de calcul / temps de chargement... À vous d'adapter la bonne technique en fonction de vos développements.

Ce mécanisme peut être également utilisé pour les objets du décor : la page contient alors des blocs qui seront positionnés pour former le décor. La page de sprites de la figure 3 contient par exemple des blocs de 32 x 32 pixels.

2. Déplacement d'un sprite

Une fois en possession de la page de sprites, on peut animer le déplacement d'un personnage. Pour cela, nous allons utiliser la balise `<canvas>` qui contiendra le sprite. Le code HTML est très simple, il suffit d'incorporer ladite balise :

```

01: <!DOCTYPE HTML>
02: <html lang="en">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test de sprites</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="script/anim.js"></script>
08:     <link rel="stylesheet" href="style.css" />
09:   </head>
10:
11:   <body>
12:     <canvas id="sprite"></canvas>
13:   </body>
14: </html>

```

Pour centrer le `canvas` horizontalement, nous aurons besoin de quelques lignes de CSS. Nous en profiterons pour changer la couleur de fond du `canvas` pour faire ressortir notre sprite :

```

01: #sprite
02: {
03:   display : block;
04:   margin-left : auto;
05:   margin-right : auto;
06:   border : 1px solid #000;
07:   background-color : #000;
08: }

```

Dans le code HTML, deux fichiers JavaScript sont utilisés : le framework jQuery (qui ne servira en fait que par habitude pour la fonction `ready()`, on aurait tout aussi bien pu utiliser `onLoad()`) et `anim.js` qui va gérer l'affichage du sprite

dans le `canvas`. Le code étant un peu long (92 lignes), je ne le présenterai pas d'un bloc et les explications ne seront pas données dans l'ordre réel des lignes (qui conserveront toutefois leur numéro originel pour plus de lisibilité).

```

60: $(document).ready(function()
61:   {
62:     var canvas_sprite = document.getElementById("sprite");
63:     var ctx_sprite = new Object(
64:       {
65:         'ctx': canvas_sprite.getContext('2d'),
66:         'width': 500,
67:         'height': 55
68:       }
69:     );
70:     var dog = new Object(
71:       {
72:         'img': new Image(),
73:         'width': 55,
74:         'height': 55,
75:         'pos': 'wait_1',
76:         'direction': 0,
77:         'sprites': {
78:           'wait_1': { 'x': 424, 'y': 4 },
79:           'wait_2': { 'x': 489, 'y': 4 },
80:           'walk_1': { 'x': 215, 'y': 4 },
81:           'walk_2': { 'x': 150, 'y': 4 },
82:         }
83:       }
84:     );
85:
86:     initCanvas(canvas_sprite, ctx_sprite);
87:     dog.img.src = 'assort.png';
88:     drawDog.pos = { 'x': 445, 'y': 0 };
89:     drawDog.anim = 0;
90:     setInterval(function() { loop(ctx_sprite, dog); }, 1000/30);
91:   }
92: );

```

Une fois la page chargée, nous récupérons le nœud de l'élément `<canvas>` identifié par `sprite` dans notre code HTML (ligne 62). Nous créons ensuite un contexte de dessin 2D pour le `canvas`. Au lieu de se contenter de ce simple objet, nous allons créer un objet contenant des informations supplémentaires : la longueur et la hauteur du `canvas` (lignes 63 à 69). Ainsi, l'objet de contexte est contenu dans `ctx_sprite.ctx`, la longueur du `canvas` dans `ctx_sprite.width`, etc.

Dans les lignes 70 à 84, nous créons un deuxième objet qui contiendra les informations sur notre sprite. Cet objet n'est pas standard, c'est simplement une solution permettant de stocker toutes les informations relatives à un sprite dans une seule variable. Les différentes clés utilisées dans la table sont :

- `img` : objet `Image` permettant d'indiquer l'image à charger ;
- `width` et `height` : longueur et hauteur du sprite. Nous avons vu que chaque image avait une taille de 55 x 55 pixels, ce seront donc les valeurs de ces clés ;
- `pos` : le nom de la position actuelle du sprite (voir clé `sprites`) ;
- `direction` : direction du sprite (`0` : attente, `-1` : vers la gauche, `1` : vers la droite). Cette clé peut être utilisée pour savoir dans quelle direction se déplace le sprite dans le cas d'un contrôle au clavier. Ici, l'animation sera imposée par le code et nous ne nous servirons pas de cette clé.

- **sprites** : tableau contenant la liste des sprites utilisés pour l'animation du chien. Chaque sprite est identifié par un nom et contient les coordonnées permettant de le récupérer sur la page de sprites. Pour connaître ces coordonnées, on peut utiliser GIMP et l'outil de sélection. Les coordonnées du point supérieur gauche de la sélection apparaissent dans la fenêtre des outils, comme le montre la figure 4.

Une fois les variables initialisées, il faut dessiner sur le **canvas**. La ligne 86 fait appel à la fonction **initCanvas()** qui va modifier la longueur et la hauteur du **canvas** en fonction des données stockées dans notre objet de contexte **ctx_sprite** :

```
01: function initCanvas(canvas, context)
02: {
03:   canvas.width = context.width;
04:   canvas.height = context.height;
05: };
```

Attention : la modification des dimensions du **canvas** ne doit pas être effectuée en CSS, mais soit directement dans le code HTML par les attributs **height** et **width**, soit en JavaScript pour éviter une déformation de vos sprites (redimensionnement de la taille des images proportionnellement à la modification de dimension du **canvas** par rapport à sa taille originale).

En ligne 87, nous indiquons la page de sprites à télécharger. Les lignes 88 et 89 sont un peu particulières. En JavaScript tout est objet, y compris les fonctions. Nous utilisons ici une fonction **drawDog()** chargée de dessiner le chien pour y stocker des données supplémentaires : la position de notre sprite sur le **canvas** (**pos**) et un compteur d'animation (**anim**). Ces variables ne seront initialisées qu'une fois, puis utilisées dans la fonction comme des variables statiques.

Pour finir, nous définissons une fonction de rappel en ligne 90. La fonction **loop()**, chargée de mettre à jour l'affichage, sera appellée toutes les 1000/30 ms. Mais pourquoi utiliser une telle écriture pour déterminer l'intervalle de temps ? Nous aurions pu noter directement une valeur arrondie, résultat du calcul, comme 34, mais nous avons vu précédemment que la vitesse d'animation était calculée en fps et comme

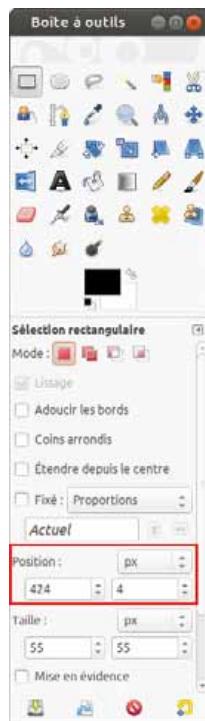


Fig. 4 : Obtention des coordonnées d'un sprite avec GIMP

1000 ms = 1 s, 1000/30 représente 30 fps. À partir de 24 images par seconde, l'œil humain ne perçoit plus la succession d'images fixes. Il faut donc un minimum de 24 fps. Ce paramètre reste très subjectif et va dépendre du joueur, du moniteur, du type de jeu, du type d'animation, etc. À régler en fonction de ces considérations ou, mieux, à rendre paramétrable.

```
54: function loop(ctx_object, image)
55: {
56:   clearCanvas(ctx_object);
57:   drawDog(ctx_object, image);
58: };
```

La fonction **loop()** effectue deux tâches : effacer le **canvas** (ligne 56) et mettre à jour l'affichage (ligne 57). La fonction **clearCanvas()** se charge de nettoyer le **canvas** :

```
07: function clearCanvas(ctx_object)
08: {
09:   ctx_object.ctx.clearRect(0, 0, ctx_object.width,
10:   ctx_object.height);
11: };
```

L'effacement du **canvas** est réalisé en effaçant une zone rectangulaire de la taille du **canvas** (ligne 9).

La fonction **drawDog()** affiche le sprite du chien dans sa nouvelle position :

```
24: function drawDog(ctx_object, image)
25: {
26:   drawDog.anim++;
27:   if ((drawDog.anim <= 120) && (drawDog.anim % 20 == 0))
28:   {
29:     image.pos = switchImage(image.pos);
30:   }
31:   else if (drawDog.anim == 121)
32:   {
33:     image.pos = 'walk_1';
34:   }
35:   else if ((drawDog.anim > 121) && (drawDog.anim < 500) && (drawDog.anim % 5 == 0))
36:   {
37:     image.pos = switchImage(image.pos);
38:     drawDog.pos.x -= 5;
39:   }
40:   else if (drawDog.anim == 500)
41:   {
42:     image.pos = 'wait_1';
43:   }
44:   else if ((drawDog.anim > 500) && (drawDog.anim % 40 == 0))
45:   {
46:     image.pos = switchImage(image.pos);
47:   }
48:   ctx_object.ctx.drawImage(image.img, image.sprites[image.pos].x,
49:                           image.sprites[image.pos].y, image.width,
50:                           image.height, drawDog.pos.x, drawDog.pos.y,
51:                           image.width, image.height);
52: };
```

On utilise le compteur d'animation **drawDog.anim** pour contrôler l'animation. La première étape est donc de l'incrémenter à chaque passage dans la fonction (ligne 26), puis de tester sa valeur pour déterminer une action.

De 0 à 120, le chien attend (lignes 27 à 30), à 121 il commence à marcher vers la gauche (lignes 31 à 34), de 121 à 499 il marche vers la gauche (lignes 35 à 39), puis à 500 il recommence à attendre (lignes 40 à 43) et ce, indéfiniment (lignes 44



Fig. 5 : Déplacement du sprite d'un chien dans un « canvas »

à 47). Tout au long de cette animation, nous utilisons la fonction `switchImage()` qui permet de permuter les images `nom_image_1` et `nom_image_2` :

```
12: function switchImage(pos)
13: {
14:   if (pos[pos.length-1] == '1')
15:   {
16:     return pos.substring(0, pos.length-1) + '2';
17:   }
18:   else
19:   {
20:     return pos.substring(0, pos.length-1) + '1';
21:   }
22: };
```

Lors du déplacement du sprite, on diminue la position d'affichage sur x de 5 px (ligne 38). Dans les tests, le modulo (%) est utilisé pour alterner les images toutes les 20, 5, ou 40 animations (lignes 27, 35, et 44). Ce mécanisme est utilisé également pour la position d'attente du chien, de manière à ce qu'il ne soit pas statique. Le résultat obtenu est présenté en figure 5.

3. Faire évoluer un sprite dans un décor

Nous savons maintenant faire avancer un personnage. Mais dans un décor ce serait plus intéressant ! Pour créer notre décor, nous aurons besoin à minima de trois `canvas` : le premier plan, le sprite et l'arrière-plan. Le schéma de la figure 6 illustre le positionnement de ces `canvas`.

Pour créer un décor pour notre chien, nous allons utiliser les sprites de la figure 3 et ajouter un sol et des arbres en premier plan, ainsi que des arbres en arrière-plan. Ainsi, nous pourrons faire marcher le chien à travers les arbres. Les `canvas` seront déclarés dans le code HTML par :

```
01: <section id="layers">
02:   <canvas id="bg" width="768px" height="440px"></canvas>
03:   <canvas id="sprite" width="768px" height="55px"></canvas>
04:   <canvas id="fg" width="768px" height="440px"></canvas>
05: </section>
```

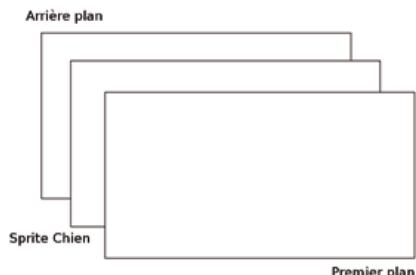


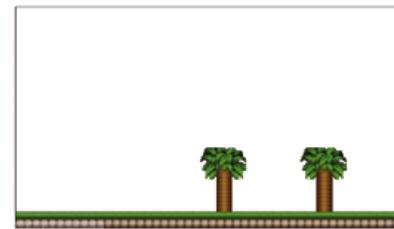
Fig. 6 : Utilisation de multiples « canvas » pour créer un décor

Fig. 7 : Contenu des « canvas » que nous allons créer

Arrière plan



Sprite



Premier plan

Fig. 8 : Écran final à obtenir par superposition des « canvas »



La taille de chacun de nos `canvas` est fixée dans ce code. Remarquez que le `canvas` du sprite n'a qu'une hauteur de 55 pixels : le chien ne se déplace que sur le sol. Une fois construits avec les différents blocs, nos `canvas bg`, `sprite` et `fg` contiendront les graphismes de la figure 7. En juxtaposant les trois `canvas` qui sont, je le rappelle, transparents, nous obtiendrons l'écran présenté en figure 8.

Pour centrer nos `canvas` et indiquer l'ordre de superposition, nous aurons besoin d'une feuille CSS un peu plus conséquente que précédemment :

```
01: #layers
02: {
03:   position: relative;
04:   margin: 0 auto;
05: }
06:
07: #layers canvas
08: {
09:   position: absolute;
10:   left: 50%;
11:   margin-left: -384px;
12: }
13:
14: #fg, #bg
15: {
16:   border: 1px solid #000;
17: }
18:
19: #sprite
20: {
21:   top: 365px;
22:   border: none;
23: }
24:
25: #fg
26: {
```

```

27: z-index: 3;
28: }
29:
30: #sprite
31: {
32: z-index: 2;
33: }
34:
35: #bg
36: {
37: z-index: 1;
38: }
```

La superposition des **canvas** est déterminée par la valeur de leur règle **z-index** (lignes 27, 32, et 37). Ici, nous avons utilisé les valeurs 1, 2 et 3, mais dans un projet plus conséquent où de nouveaux **canvas** risquent d'être intercalés, préférez des valeurs supérieures laissant de la place disponible (par exemple 100, 200 et 300).

Le code permettant d'afficher ces blocs sera sensiblement le même que celui utilisé pour afficher le chien... mais sans animation, bien sûr !

```

01: var canvas_fg = document.getElementById('fg');
02: var ctx_fg   = canvas_fg.getContext('2d');
03: var canvas_bg = document.getElementById('bg');
04: var ctx_bg   = canvas_bg.getContext('2d');
05:
06: var blocks   = new Object(
07:   {
08:     'img': new Image(),
09:     'width': 32,
10:     'height': 32,
11:     'sprites': {
12:       'floor_1': { 'x': 104, 'y': 1},
13:       'trunk_1': { 'x': 240, 'y': 137},
14:       'leaves_1': { 'x': 240, 'y': 69},
15:       'leaves_2': { 'x': 240, 'y': 35},
16:       'leaves_3': { 'x': 206, 'y': 69},
17:       'leaves_4': { 'x': 206, 'y': 35},
18:       'leaves_5': { 'x': 274, 'y': 69},
19:       'leaves_6': { 'x': 274, 'y': 35},
20:       'trunk2_1': { 'x': 342, 'y': 103},
21:       'trunk2_2': { 'x': 342, 'y': 137},
22:       'leaves2_1': { 'x': 342, 'y': 69},
23:       'leaves2_2': { 'x': 342, 'y': 35},
24:       'leaves2_3': { 'x': 308, 'y': 69},
25:       'leaves2_4': { 'x': 308, 'y': 35},
26:       'leaves2_5': { 'x': 376, 'y': 69},
27:       'leaves2_6': { 'x': 376, 'y': 35},
28:     }
29:   );
30:   blocks.img.src = 'blocks2.png';
31:
```

On retrouve la structure permettant de définir le positionnement de chaque sprite dans les lignes 6 à 30. L'affichage des **canvas** d'arrière-plan et de premier plan se fera par appel de deux fonctions **drawBg()** et **drawFg()** ajoutées à la fonction de rappel :

```

01: setInterval(function()
02:   {
03:     loop(ctx_sprite, dog);
04:     drawBg(ctx_bg, blocks);
05:     drawFg(ctx_fg, blocks);
06:   },
07:   1000/30);
```

Ces deux **canvas** étant fixes, pourquoi les intégrer à la boucle ? Cela permet de s'assurer de leur affichage sur la page et il pourrait être intéressant d'y ajouter par la suite des objets mouvants (nuages dans le ciel pour l'arrière-plan, herbes et insectes pour le premier plan, etc.). Les fonctions d'affichage sont très simples et je ne présenterai que la première, les différences entre les deux se faisant au niveau des objets affichés et de leur coordonnées.

```

01: function drawFg(context, image)
02: {
03:   // Sol
04:   for (var i=0; i<768; i+=32)
05:   {
06:     context.drawImage(image.img, image.sprites['floor_1'].x,
07:                      image.sprites['floor_1'].y, image.width,
08:                      image.height, i, 408,
09:                      image.width, image.height);
10:   }
11:
12:   // 2 arbres
13:   for (var j=0; j<=200; j += 200)
14:   {
15:     // Tronc
16:     for (var i=0; i<=32; i+=32)
17:     {
18:       context.drawImage(image.img, image.sprites['trunk_1'].x,
19:                         image.sprites['trunk_1'].y, image.width,
20:                         image.height, 600-j, 376-i,
21:                         image.width, image.height);
22:     }
23:
24:   // Feuilles
25:   context.drawImage(image.img, image.sprites['leaves_1'].x,
26:                     image.sprites['leaves_1'].y, image.width,
27:                     image.height, 600-j, 312,
28:                     image.width, image.height);
29:   // ...
30:   context.drawImage(image.img, image.sprites['leaves_6'].x,
31:                     image.sprites['leaves_6'].y, image.width,
32:                     image.height, 632-j, 280,
33:                     image.width, image.height);
34:   }
35: }
```

Ce code a été raccourci, mais vous pouvez voir une forte redondance de code avec de multiples appels à la fonction **drawImage()** dans les lignes 6, 18, 25, les lignes omises en 29, et la ligne 30. C'est là que l'on voit l'intérêt de travailler sur un éditeur de niveaux...

4. Conclusion

Grâce aux **canvas**, nous pouvons facilement mettre en place l'utilisation de sprites pour réaliser des écrans de jeu statiques ou animés. Un seul point peut s'avérer rapidement bloquant : l'organisation du code. Avec la multiplication des sprites utilisés pour le décor et pour les personnages joueurs et non joueurs, les programmes risquent de devenir rapidement illisibles ! Pensez à factoriser votre code, à créer un éditeur de niveaux et à séparer vos fichiers... Vous pouvez tout à fait créer un fichier JavaScript par page de sprites, puis écrire un générateur de fichiers JavaScript qui regroupera tous vos fichiers en un seul pour éviter de télécharger de multiples petits fichiers. Donc, même si vous êtes enthousiaste après vos tests sur les sprites, prenez le temps de la réflexion avant de vous lancer dans un développement plus complexe... ■



HTML 5 ET SVG

par Cédric Gémy

Le SVG, qui a plus de 10 ans, était presque inconnu sauf de certains que l'on regardait d'un œil retors. Et voilà pourtant qu'il prend vie grâce aux évolutions du HTML. Un avenir radieux se présenterait-il pour une nouvelle forme de graphisme web ?



1. Introduction

Le format SVG (*Scalable Vector Graphics*) est depuis peu mis en avant, disons depuis l'arrivée d'HTML 5. SVG est considéré comme une nouvelle technologie, une sorte de complément graphique moins complexe que **canvas**. En fait, SVG est une technologie beaucoup plus ancienne, plus ou moins restée dans l'ombre lors de son apparition à cause du refus de Microsoft de le supporter (Internet Explorer était alors majoritaire) et de la place prépondérante de Flash. D'autres raisons peuvent aussi être invoquées. Par exemple, SVG était certainement en avance sur son temps : scriptable, animable, des tours de passe-passe un peu complexes pour des graphistes et intégrateurs web de la première, ont fait que l'attention ne s'y est pas portée. Les graphistes de l'époque (il y en a encore beaucoup trop maintenant) regardent le vectoriel avec des yeux méfiants et les développeurs n'avaient pas les briques et les ressources réseau pour exploiter son potentiel. Mais les choses ont changé en grande partie.

Amateurs de logiciels libres, nous avons une chance, car les choix stratégiques de valoriser les normes et recommandations ouvertes ont fait que ces standards s'imposent et que la plupart des outils qui existent à l'heure actuelle pour manipuler le SVG sont libres, en particulier le logiciel Inkscape. Les bibliothèques permettant de le manipuler à la volée ont aussi évolué, ouvrant un potentiel

énorme pour les graphiques, diagrammes, cartes géographiques interactives animées ou non, éventuellement les jeux. Le groupe de travail HTML 5 a donc concentré des efforts à mieux intégrer SVG aux technologies de son environnement comme CSS, ECMAScript et fournir un ensemble plus cohérent...

2. Découverte du SVG

Le SVG (<http://www.w3.org/TR/SVG11/>) est un langage de la famille XML, facile à prendre en main. Son contenu aura en particulier l'avantage d'être indexable, ce qui le rend sans équivalent par rapport à des formats binaires comme Flash, ou bitmap comme **canvas**. Comme son nom l'indique, un dessin SVG est étirable (ou zoomable) sans pertes, ce qui le rend parfait pour la conception de documents qui doivent s'afficher sur des gammes de périphériques très hétérogènes allant du smartphone à l'écran 30 pouces.



Fig. 1

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
  width="67"
  height="67"
  id="svg3199"
  version="1.1">
<g
  id="layer1"
  transform="translate(-103.57143,-255.93361)">
<g
  transform="translate(-142.85715,-232.14286)"
  id="ico_dont">
<path
  transform="matrix(1.46875,0,0,1.46875,40.803571,213.49128)"
  d="m 185.71429,209.80876 c 0,12.62365 -10.23349,22.85714 -22.85714,22.85714
  150.2335,232.6659 140,222.43241 140,209.80876 c 0,-12.62365 10.2335,-22.85714 22.85714,-22.85714
  12.62365,0 22.85714,10.23349 22.85714,22.85714 z"
```

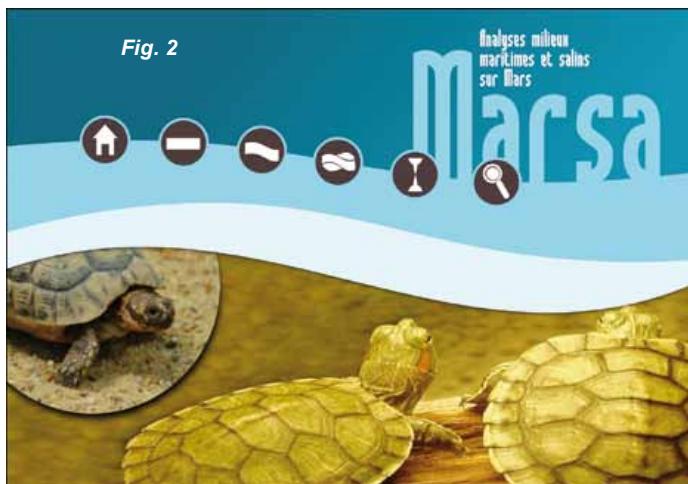
```

    ry="22.857143"
    rx="22.857143"
    cy="209.80876"
    cx="162.85715"
    id="path2992"
    style="fill:#483737" />
<rect
    style="fill:#ffffff;fill-opacity:1;stroke:none"
    id="btn_dont"
    width="50.357143"
    height="16.785715"
    x="254.28571"
    y="514.14789" />
</g>
</g>
</svg>

```

L'exemple de la figure 1, simplifié d'un fichier Inkscape représentant un panneau stop, montre les capacités de dessin par formes géométriques (ici `rect`), un chemin vectoriel (qui serait ici le cercle du panneau représenté par `path` avec une ligne tracée en `d` selon les coordonnées fournies pour chaque vecteur). On peut aussi observer le mix entre attributs de type XML et les expressions CSS, ainsi que la possibilité de faire des transformations géométriques.

En poussant plus loin, il ne reste que l'imagination et des pratiques pour intégrer le dessin dans tous les types de produit qui le nécessitent. En figure 2, une illustration représentant une interface de navigation simple incluant de nombreuses formes courbes et des compositions non linéaires avec des masques, des flous le long des formes, etc.



3. Intégration dans HTML 5

3.1 Insertion dans la page

`Embed`, `object` ou autre, fini les vieilles questions ! Même si ces méthodes sont toujours possibles, SVG fait maintenant partie intégrante du HTML 5. Il est donc possible de placer directement le code SVG dans la page. Il n'est même pas nécessaire de définir l'espace de nom.

```

<!DOCTYPE html>
<html>
  <head>
    <title>HTML5 SVG demo</title>
  </head>

  <body>
    <svg id="circle" height="220" xmlns="http://www.w3.org/2000/svg">
      <circle id="cercle" cx="100" cy="100" r="50" fill="red" />
    </svg>
  </body>
</html>

```

Le recours à une ressource externe pouvant cependant s'avérer plus judicieuse, il est à présent possible d'utiliser un simple fichier SVG en valeur d'attribut d'une image :

```

<p> Il est possible d'écrire du texte et de charger le fichier SVG complet comme une image

</p>

```

Ou encore en arrière-plan à l'aide des propriétés CSS `background-image` et associées :

```

<style type="text/css">
  body {
    background-image: none, url('cercle_rouge.svg');
    background-position: 50% 50%;
    background-repeat: no-repeat;
  }
</style>

```

L'avantage de l'arrière-plan SVG tient tout simplement dans le fait qu'il peut être mis à l'échelle comme le reste de la page, voire animé si besoin et éventuellement plus léger. On peut même, si on le souhaite, intégrer plusieurs arrière-plans grâce aux nouvelles propriétés CSS 3.

3.2 Interactions HTML/SVG

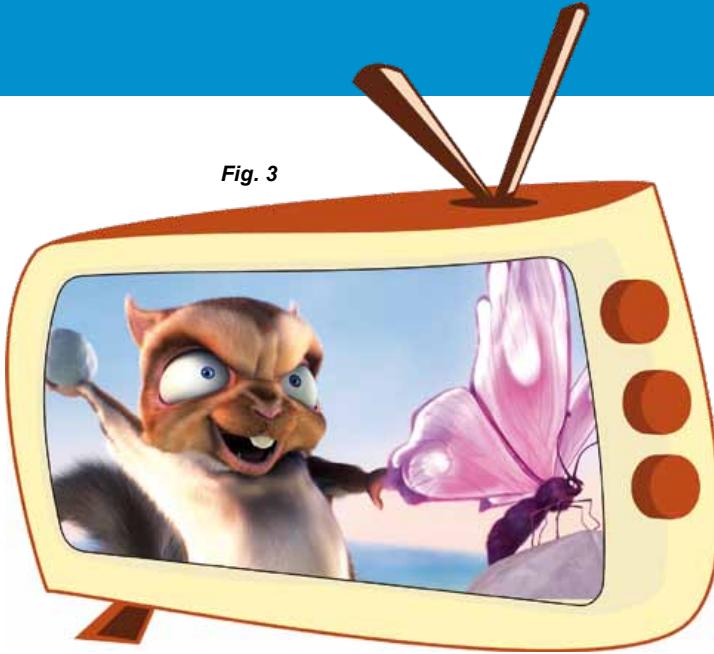
Maintenant, le SVG n'est pas qu'un simple composant graphique qui viendrait remplir la page. HTML 5 peut maintenant accéder aux objets définis dans un fichier SVG pour les utiliser sur son propre contenu. Parmi ceux-ci, on retrouve l'interaction avec les nouveaux objets audio/vidéo, les effets, les fontes.

3.2.1 Masques et interface

SVG est un langage parfait pour la composition graphique de l'interface. De fait, la quasi-totalité des icônes sont maintenant réalisées dans ce format. Rien de plus étrange qu'il soit possible de créer des contrôleurs audio et vidéo en SVG et de faire interagir les 2.

Enfin, le SVG peut être utilisé comme masque pour cacher des parties de la vidéo qui sont immanquablement rectangulaires jusqu'alors. Dans l'exemple simplifié ci-dessous, un style permet de définir une image décorative et un second le masque à utiliser. Dans ce cas, il s'agit du même dessin que nous exploitons en faisant appel au bon moment à un objet masque spécifique inclus, défini dans ce fichier.

Fig. 3



```
<style>
    .video_bg{
        background-image:url("dessin.svg");
        background-repeat:no-repeat;
    }
    .target {
        mask: url("dessin.svg#mask");
    }
</style>
```

Il suffit alors d'y faire référence en tant que classe :

```
<div class="video_bg">
    <video class="target" height="270" width="500" controls >
        <source src="sintel-1024-stereo-small.mp4" type="video/mp4" />
        <source src="sintel.ogv" type="video/ogg" />
    </video>
</div>
```

Dans un monde parfait, il devrait être possible de se passer du **div**, mais les arrière-plans sur l'élément **video** ne semblent pas implémentés à l'heure actuelle. Dans notre cas, voir en figure 3 le genre de résultat obtenu.

3.2.2 Effets

Dans la même veine, les primitives de filtres SVG, qui servent à créer des effets spéciaux, sont exploitables dans une page HTML 5 si celle-ci contient une instruction SVG qui les définit au préalable. Les primitives ne sont cependant pas la partie la plus simple de SVG, malgré l'effort de certains pour en simplifier l'accès.

Note

Inkscape possède un menu listant des filtres préconçus et classés par catégories, ainsi qu'une interface qui facilite la création d'effets à partir de filtres SVG, l'**« éditeur de filtres »**. Ce dernier mériterait un rafraîchissement, mais il n'en reste pas moins intéressant.

3.2.3 Fontes

Les fontes SVG sont des possibilités encore très peu utilisées et qui ne sont pas nécessairement destinées à une grande audience. Les fontes SVG possèdent cependant plusieurs avantages : elles peuvent être colorées, utiliser des effets (comme du flou) en natif, être utilisées dans le document même qui les définit. Dans le cas d'un document scripté, le fait d'avoir accès aux données composant les glyphes dans une forme claire permet d'agir facilement sur l'aspect du texte en cas de besoin et de créer d'éventuels graphiques sur le texte à la volée, comme des déformations.

L'intégration de fontes SVG dans un document HTML 5 se fait de la même façon qu'avec les autres polices de type TTF ou OTF, avec la propriété **@font-face**. Sur ce point, voir l'article « Utiliser des polices externes », en page 44 du présent numéro.

Autre point méconnu du logiciel Inkscape, il possède aussi une interface graphique facilitant la création de fontes, y compris SVG et s'offre comme un bon complément à FontForge. Pour plus d'infos à ce sujet, parcourez la documentation écrite par l'équipe de FLOSS Manuals : <http://fr.flossmanuals.net/fontes-libres/> ou <http://www.flossmanualsfr.net>.

4. Intégration avec CSS

En ce qui concerne l'intégration avec CSS, nous avons déjà parlé de la possibilité de mettre un fichier SVG en arrière-plan. Mais l'intégration CSS et SVG va beaucoup plus loin. En fait, les dernières avancées du CSS 3 reprennent de nombreuses fonctionnalités du SVG : couleurs exprimées en RGBA, ombre portée, angles arrondis, animation, transformations... Et cela semble continuer puisque les derniers *Working Drafts* du mois d'août font passer les filtres SVG et les modes de fusion au rang d'éligibles en CSS. Bref, SVG était bien en avance sur son temps et le fait que ces technologies soient récupérées le prouve et va rendre plus facile son exploitation, en particulier en termes graphiques (dessin en particulier).

Dans tous les cas, le fichier SVG peut partager le fichier CSS avec la page web, ce qui facilite l'intégration du dessin, d'un graphique ou d'une carte au thème choisi.

5. Intégration via des scripts

5.1 Technologies standards

En tant que technologie web, le SVG peut aussi interagir avec le contenu web par le biais de scripts. Initialement publié avec des spécificités, SVG a peu à peu été rattrapé par les technologies maintenant devenues courantes comme jQuery.

Le plugin jQuery SVG permet un accès facilité au DOM SVG et de manipuler plus facilement les objets qu'avec la librairie classique.

Rappelons qu'en tant que technologie XML, SVG peut être traité avec Xpath, XSLT ou encore être généré directement par tout langage serveur, comme PHP ou Python. Mais si cela est rare et n'est pas un objectif en soi, on pourrait imaginer un site entièrement construit en SVG, comme c'est le cas pour certains diaporamas disponibles et produits à l'aide d'extensions Inkscape comme JessyLink (<http://code.google.com/p/jessylink/>) ou Sozi (<http://sozi.baierouge.fr/wiki/sozi>). Et cela est déjà disponible. On voit bien à quel point SVG a fait son chemin petit à petit et qu'il arrive à présent à une maturité d'utilisation qui donne des envies.

CSS et éventuellement une petite dose de JavaScript pour contrôler l'interaction graphique, soit dans l'image elle-même, soit via un contrôleur placé dans la page web (Fig. 4).

5.2 Utiliser des librairies spécialisées

Il existe de nombreuses librairies basées sur JavaScript permettant de générer du SVG côté client. Parmi celles-ci, Raphaël est certainement la plus connue. L'utilisation de telles librairies n'est pas évidente du point de vue du graphiste, mais se comprend immédiatement du point de vue du dynamisme. Les scripts permettront de générer les SVG à la volée, selon une source de données, qui peut, le cas échéant, être couplée avec un base SVG produite par un dessinateur. Raphaël, en particulier grâce à son extension



gRaphaël (<http://g.raphaeljs.com/>), se destine plus particulièrement aux diaporamas et graphiques, en vogue à l'heure actuelle dans les outils d'analyse.

Mais le mouvement vers l'ouverture des données fait que le regard se porte aussi sur les possibilités offertes en termes de cartographie. D'ailleurs, OpenStreetMap utilise depuis longtemps SVG, preuve de la pérennité du format et de sa fiabilité. L'interaction entre les données du serveur, ou de la page web (tableau, saisie utilisateur, liste...) peut alors être maximale sans utilisation de technologie privative (comme dans l'exemple tiré de Raphaeljs.com, en figure 5).

le W3C n'ait pour l'instant fait que mettre l'existant au goût du jour en espérant que la sauce, cette fois, prendra.

Si c'est le cas, SVG n'en restera pas moins une vraie technologie de dessin, qui s'engage lui-même vers des voies pré-presse ou proches des métiers de la création graphique. Il en devient un véritable concurrent à feu EPS et reste le format vectoriel supporté dans le format ePub, lui-même basé sur HTML 5 et connaissant l'explosion actuelle du secteur, l'avenir s'annonce riche.

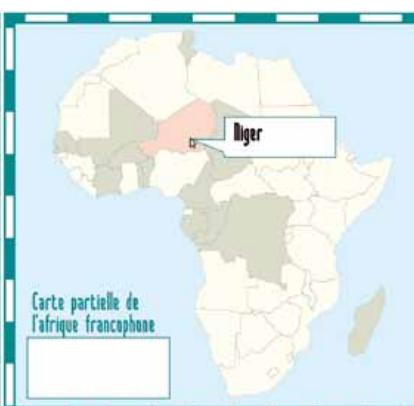
Pour avoir des exemples *live* des possibilités listées dans cet article, rendez-vous sur <http://www.cgemy.com?p=Graphisme.SVG.exemples>. ■

6. L'avenir

Après avoir répété que SVG était pionnier sur de nombreux points et que son intégration se fait par absorption progressive, cela va-t-il signifier qu'utiliser SVG devient inutile ? Il est évident que non. Du moins, dans un premier temps. Les graphistes, pour une partie, sont des personnes qui ont souvent peur de la moindre ligne de code et ne s'y mettent que par obligation ou contrainte. Le manque d'outils d'autorising SVG complets a fait que cette recommandation est restée dans l'ombre et que

Note

Les navigateurs. La vraie question est de savoir comment SVG est supporté par les navigateurs ? Si ce ne fut pas toujours le cas, les choses vont très bien maintenant, puisque même les navigateurs propriétaires, à la traîne, s'y sont faits. Il reste toujours des parties non supportées ici ou là, mais l'essentiel y est. Référez-vous au site suivant pour vérifier : <http://caniuse.com/#cats=SVG>.





LES CONTRÔLES AU CLAVIER ET À LA SOURIS

par Tristan Colombo

L

'interaction avec le joueur se fait essentiellement à l'aide de deux périphériques : le clavier et la souris. Comment traiter les informations envoyées par le joueur ? C'est le but de cet article...



Dans un jeu, on utilise une boucle pour scruter les différents événements :

1. Présentation des données du jeu à l'utilisateur : en fonction du jeu, affichage du plateau de jeu et des pions, affichage des cartes, etc. ;
2. Le joueur indique une action à effectuer : clic sur une zone avec la souris, appui sur une touche du clavier, mouvement devant une caméra, etc. ;
3. Résolution de l'action : adaptation du jeu (affichage, données) à l'action définie par le joueur ;
4. Retour à l'étape 1.

Dans cet article, nous nous concentrerons sur la phase 2 et plus particulièrement sur l'interception et la gestion des événements du clavier et de la souris. Dans un premier temps, nous étudierons la gestion du clavier en JavaScript pur pour bien en comprendre le fonctionnement. Dans un second temps, nous utiliserons le framework jQuery pour simplifier notre code et nous utiliserons ces techniques pour déplacer un *sprite* (voir l'article « Animer un sprite » du présent magazine). Enfin, nous aborderons la gestion des événements souris et plus particulièrement le glisser-déposer ou *drag and drop*.

1. La gestion des événements clavier en JavaScript

La chose la plus importante à retenir lorsque l'on veut gérer des événements clavier est qu'il n'existe pas de norme définissant les raccourcis des navigateurs web. Il faudra donc faire très attention aux combinaisons de touches ([Ctrl]+[Q] par exemple), qui seront peut-être interceptées par le navigateur avant que vous n'ayez pu les traiter.

Ensuite, pour récupérer une touche tapée au clavier, il suffira d'écouter les événements `keypress`, `keydown` ou `keyup`. Les deux premiers événements sont similaires et

sont déclenchés lors de l'appui sur une touche. L'événement `keyup` est déclenché lorsque la touche est relâchée (donc avec un temps de retard).

Chaque touche est identifiée par un code : il s'agit du code ASCII (*American Standard Code for Information Interchange*). Pour connaître le code d'une lettre, il faut alors rechercher dans la table ASCII la correspondance. De nombreuses tables sont disponibles sur Internet [1] et vous simplifieront le travail. La flèche vers le haut a par exemple pour code la valeur 38.

Pour afficher un message dans la console de Firebug en cas d'appui sur une touche, nous devrons taper le code suivant :

```
01: document.onkeydown = function(event)
02: {
03:   console.log(event.which + ' : ' + String.fromCharCode(event.which));
04: }
```

La surveillance de l'événement `keydown` est activée en ligne 1 et en ligne 3 nous indiquons que nous souhaitons afficher le code de la touche (attribut `which` de l'objet `event`) et la conversion de ce code en lettre par la fonction `String.fromCharCode()`.

Le traitement de ces différents codes n'est pas très pratique ! Imaginez la dégradation de la lisibilité de votre code lorsque vous faites des tests sur les valeurs 37 à 40 pour détecter l'appui sur l'une des quatre flèches ! Comme bien souvent en JavaScript, jQuery va venir à notre secours.

2. Simplification du contrôle au clavier avec l'incontournable jQuery

L'extension Hotkeys [2] pour jQuery va nous permettre de nommer les touches. Ainsi, pour [Ctrl]+[S], nous pourrons utiliser `ctrl_s`, pour la flèche gauche nous pourrons

utiliser `left`, etc. La fonction `bind()` de jQuery nous permettra d'activer la surveillance des événements de manière classique.

Voici un exemple permettant de déplacer un simple `div` que nous colorierons en rouge. Le code HTML est très simple :

```
01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test sur le clavier</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="script/jquery.hotkeys.js"></script>
08:     <script src="script/keyboard.js"></script>
09:     <link rel="stylesheet" href="style.css" />
10:   </head>
11:   <body>
12:     <div id="block"></div>
13:   </body>
14: </html>
```

Ici, il faut simplement ne pas oublier de charger la librairie jQuery en ligne 6, puis l'extension Hotkeys en ligne 7. Le carré que nous allons déplacer à l'aide des flèches se nomme « `block` » (ligne 12). Pour le colorier en rouge et lui donner une taille de départ, nous écrirons quelques règles CSS dans le fichier `style.css` qui est chargé par la ligne 9 :

```
01: #block
02: {
03:   position: absolute;
04:   width: 20px;
05:   height: 20px;
06:   background-color: #f00;
07: }
```

Enfin, la partie la plus importante avec le contrôle du clavier et des mouvements du carré rouge depuis le JavaScript :

```
01: $(document).ready(function()
02: {
03:   $(document).bind('keydown.left', function(event)
04:   {
05:     var val = parseInt($('#block').css('left')) - 1;
06:     $('#block').css('left', val + 'px');
07:   });
08:   $(document).bind('keydown.right', function(event)
09:   {
10:     var val = parseInt($('#block').css('left')) + 1;
11:     $('#block').css('left', val + 'px');
12:   });
13:   $(document).bind('keydown.up', function(event)
14:   {
15:     var val = parseInt($('#block').css('top')) - 1;
16:     $('#block').css('top', val + 'px');
17:   });
18:   $(document).bind('keydown.down', function(event)
```

```
22:   {
23:     var val = parseInt($('#block').css('top')) + 1;
24:     $('#block').css('top', val + 'px');
25:   }
26: }
27: }
28: );
```

À l'activation de la surveillance de chaque événement, nous concaténons le nom de la touche au nom de l'événement. Ainsi, en ligne 3, `keydown.left` permet de détecter l'appui sur la touche flèche gauche.

Pour déplacer le carré, nous récupérons à chaque fois la valeur de la règle CSS `left` ou `top` (en fonction du déplacement gauche/droite ou haut/bas) et nous incrémentons ou décrémentons cette valeur. Ce sont les lignes 5, 6 (déplacement à gauche), 11, 12 (droite), 17, 18 (haut), et 23, 24 (bas) qui effectuent cette tâche.

Dans la gestion des événements clavier associés à un jeu, cette architecture peut devenir rapidement très lourde à maintenir. Il serait plus simple de pouvoir tester depuis la boucle principale si une touche a été utilisée. Il suffit pour cela de charger un petit script supplémentaire, baptisé « `key_status` » et présenté par Daniel Moore sur le site html5rocks.com :

```
01: $(function()
02: {
03:   window.keydown = {};
04:
05:   function keyName(event)
06:   {
07:     return jQuery.hotkeys.specialKeys[event.which] ||
08:           String.fromCharCode(event.which).toLowerCase();
09:   }
10:
11:   $(document).bind("keydown", function(event)
12:   {
13:     keydown[keyName(event)] = true;
14:   });
15:
16:   $(document).bind("keyup", function(event)
17:   {
18:     keydown[keyName(event)] = false;
19:   });
20:
21: }
22: );
23: );
```

Ce script crée un objet global `keydown` qui contiendra pour chaque touche (au sens de l'extension Hotkeys) une valeur booléenne indiquant si la touche est utilisée ou non. La fonction `keyName()` des lignes 5 à 9 permet de renvoyer le nom de la touche et les lignes 11 à 15 et 17 à 21 activent la surveillance du clavier et mettent à jour les attributs de l'objet `keydown`. Désormais, pour tester si le joueur appuie sur la flèche vers la gauche, il suffira de tester la valeur de la variable `keydown.left`.

Pour mettre en place cette technique, je vous propose de l'appliquer au déplacement du sprite « chien » que nous avions créé dans l'article traitant de l'utilisation des sprites.

3. Déplacement d'un sprite

Au niveau du code HTML, nous allons récupérer l'en-tête utilisé précédemment et ajouter le chargement de `key_status.js`. La balise `<canvas>` permettant d'afficher le sprite sera le seul élément du corps de ce document.

```
01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test sur le clavier</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="script/jquery.hotkeys.js"></script>
08:     <script src="script/key_status.js"></script>
09:     <script src="script/keyboard.js"></script>
10:     <link rel="stylesheet" href="style.css" />
11:   </head>
12:   <body>
13:     <canvas id="sprite"></canvas>
14:   </body>
15: </html>
```

Le fichier `keyboard.js` reprendra globalement le code employé pour animer le sprite de manière automatique. Les modifications du code apparaissent en rouge :

```
01: function initCanvas(canvas, context)
02: {
03:   canvas.width = context.width;
04:   canvas.height = context.height;
05: };
06:
07: function clearCanvas(ctx_object)
08: {
09:   ctx_object.ctx.clearRect(0, 0, ctx_object.width, ctx_object.height);
10: };
11:
12: function drawDog(ctx_object, image)
13: {
14:   drawDog.anim++;
15:   ctx_object.ctx.drawImage(image.img, image.sprites[image.pos].x,
16:                           image.sprites[image.pos].y, image.width,
17:                           image.height, drawDog.pos.x, drawDog.pos.y,
18:                           image.width, image.height);
19: };
20:
21: function loop(ctx_object, image)
22: {
23:   if (keydown.right)
24:   {
25:     if (drawDog.anim % 5 == 0)
26:     {
27:       image.pos = image.transition[image.pos].right;
28:       if (drawDog.pos.x <= 440)
29:       {
30:         drawDog.pos.x += 5;
```

```
31:       }
32:     }
33:   }
34:   else if (keydown.left)
35:   {
36:     if (drawDog.anim % 5 == 0)
37:     {
38:       image.pos = image.transition[image.pos].left;
39:       if (drawDog.pos.x >= 5)
40:       {
41:         drawDog.pos.x -= 5;
42:       }
43:     }
44:   }
45:   else
46:   {
47:     if (drawDog.anim % 30 == 0)
48:     {
49:       image.pos = image.transition[image.pos].none;
50:     }
51:   }
52:
53:   clearCanvas(ctx_object);
54:   drawDog(ctx_object, image);
55: };
56:
57: $(document).ready(function()
58: {
59:   var canvas_sprite = document.getElementById("sprite");
60:   var ctx_sprite = new Object(
61:   {
62:     'ctx': canvas_sprite.getContext('2d'),
63:     'width': 500,
64:     'height': 55
65:   }
66: );
67:   var dog = new Object(
68:   {
69:     'img': new Image(),
70:     'width': 55,
71:     'height': 55,
72:     'pos': 'wait_1',
73:     'direction': 0,
74:     'sprites': {
75:       'wait_1': { 'x': 424, 'y': 4 },
76:       'wait_2': { 'x': 489, 'y': 4 },
77:       'walk_1': { 'x': 215, 'y': 4 },
78:       'walk_2': { 'x': 150, 'y': 4 },
79:       'walk_3': { 'x': 85, 'y': 4 },
80:       'walk_4': { 'x': 20, 'y': 4 },
81:     },
82:     'transition': {
83:       'wait_1': { 'left': 'walk_1', 'right': 'walk_3', 'none': 'wait_2' },
84:       'wait_2': { 'left': 'walk_1', 'right': 'walk_3', 'none': 'wait_1' },
85:       'walk_1': { 'left': 'walk_2', 'right': 'walk_3', 'none': 'wait_1' },
86:       'walk_2': { 'left': 'walk_1', 'right': 'walk_3', 'none': 'wait_1' },
87:       'walk_3': { 'left': 'walk_1', 'right': 'walk_4', 'none': 'wait_1' },
88:       'walk_4': { 'left': 'walk_1', 'right': 'walk_3', 'none': 'wait_1' }
```

```

89:      }
90:    }
91:  );
92:
93:  initCanvas(canvas_sprite, ctx_sprite);
94:  dog.img.src = 'assort.png';
95:  drawDog.pos = { 'x': 445, 'y': 0 };
96:  drawDog.anim = {};
97:  setInterval(function() { loop(ctx_sprite, dog); }, 1000/30);
98: }
99: );

```

Nous avons ajouté les sprites permettant au chien de se déplacer vers la droite et leurs coordonnées sont stockées dans les lignes 79 et 80.

Nous avons également un attribut supplémentaire nommé **transition** (lignes 82 à 89), qui va nous permettre de nous passer de la fonction **switchImage()** que nous avions développée précédemment : en fonction d'une position de sprite et d'un mouvement donné, nous stockons la référence au sprite suivant. Ainsi, si nous sommes en attente sur le sprite **wait_1** et qu'aucun événement clavier n'intervient, nous afficherons le sprite étiqueté par **none**, soit **wait_2**.

La boucle principale gérée par la fonction **loop()** a beaucoup changé. Nous testons si le joueur appuie sur la flèche droite (ligne 23) ou la flèche gauche (ligne 34). Si c'est le cas, nous récupérons le sprite suivant à afficher (ligne 27) et si le déplacement n'envoie pas le chien à l'extérieur du **canvas**, nous calculons ses nouvelles coordonnées (lignes 28 à 31). S'il n'y a pas de déplacement, alors nous affichons les sprites d'attente (ligne 62).

La fonction d'affichage du chien **drawDog()** a été extrêmement raccourcie puisque désormais elle se contente de compter les animations (ligne 14) et d'afficher sur le **canvas** le sprite (lignes 15 à 18).

Il est possible d'apporter une légère amélioration à ce script en gérant l'accélération : si vous vous déplacez de manière continue dans une même direction, le personnage accélérera. Ces modifications portent sur la fonction **loop()** :

```

01: function loop(ctx_object, image)
02: {
03:   // ...
04:   else if (keydown.left)
05:   {
06:     if ((drawDog.anim % 5 == 0) || (drawDog.anim % loop.fast == 0))
07:     {
08:       if ((image.pos == 'walk_1') || (image.pos == 'walk_2'))
09:       {
10:         loop.acce++;
11:       }
12:     else
13:     {
14:       loop.accel = 0;
15:       loop.fast = undefined;

```

```

16:     }
17:     if (loop.accel == 10)
18:     {
19:       loop.fast = 3;
20:     }
21:     image.pos = image.transition[image.pos].left;
22:     if (drawDog.pos.x >= 5)
23:     {
24:       drawDog.pos.x -= 5;
25:     }
26:   }
27: }
28: // ...
29: };

```

Si le joueur déplace le chien plus de dix fois consécutives dans la même direction (ligne 17), nous activons l'accélération (ligne 19). Le décompte des passages dans la boucle avec une même direction est effectué dans les lignes 8 à 16.

Ici, nous avons seulement accéléré la cadence d'affichage du sprite, mais nous aurions pu également avoir des sprites de course où les enjambées du personnage auraient été plus importantes, ce qui aurait produit un meilleur rendu.

4. La gestion des événements souris

Il n'y a pas que le clavier pour contrôler les éléments d'un jeu, la souris peut également être utilisée. La gestion de la souris avec les **canvas** a déjà été vue dans l'article sur les **canvas**. Je vous propose ici de nous intéresser au glisser-déposer dont le codage a été simplifié par HTML 5.

Le glisser-déposer est la faculté de pouvoir cliquer sur des éléments de vos pages web pour les déposer dans une zone prévue à cet effet. HTML 5 introduit de nouveaux événements, dont trois nous intéresseront particulièrement ici :

- **ondragstart** : événement déclenché lorsque l'on clique sur un objet et que l'on commence à le déplacer. Pour pouvoir être déplacé, un objet doit avoir été identifié comme élément pouvant participer au glisser-déposer grâce à l'attribut **draggable="true"** ;
- **ondragover** : événement déclenché lorsqu'un élément en cours de déplacement passe au-dessus d'une zone sur laquelle on peut le déposer ;
- **ondrop** : événement déclenché lorsque l'on lâche l'élément dans une zone pouvant l'accueillir.

Pouvoir gérer les événements liés au glisser-déposer ne suffit pas. Il faut également pouvoir transférer des informations de la zone de départ à la zone cible. Pour cela, HTML 5 nous permet d'utiliser l'objet **event.dataTransfer** qui permet de spécifier le type de déplacement (copie, déplacement, lien, etc.) par l'intermédiaire de l'attribut

`effectAllowed` et, grâce à la méthode `setData()`, nous pouvons transmettre des données. La fonction `setData()` requiert deux paramètres : le type de données qui est transmis ("Text" pour une chaîne de caractères), puis la donnée proprement dite.

Voici un petit exemple mettant en œuvre ces différentes notions :

```

01: <!DOCTYPE html>
02:
03: <html lang="fr">
04:   <head>
05:     <style>
06:       .cible
07:       {
08:         float: left;
09:         width: 200px;
10:         height: 200px;
11:         margin: 5px;
12:         border: 2px solid #aaa;
13:       }
14:
15:       .element
16:       {
17:         width: 100px;
18:         height: 100px;
19:         margin: 10px;
20:         padding: 5px;
21:         background-color: #aaa;
22:       }
23:     </style>
24:
25:   <script>
26:     function dragStart(event)
27:     {
28:       console.log('Detection de debut de drag and drop');
29:       event.dataTransfer.effectAllowed = 'move';
30:       event.dataTransfer.setData("Text", event.target.
getAttribute('id'));
31:     };
32:
33:     function dragOver(event)
34:     {
35:       console.log('Zone de largage "' + event.target.
getAttribute('id') +'" detectee');
36:       return false;
37:     };
38:
39:     function drop(event)
40:     {
41:       var element = event.dataTransfer.getData("Text");
42:       var cible_id = event.target.getAttribute('id');
43:       console.log('Largage de l\'element sur "' + cible_id +'"');
44:       if (cible_id == 'elt')
45:       {
46:         console.log('Interdiction de larguer l\'element sur lui-
meme');

```

```

47:       }
48:     else
49:     {
50:       event.target.appendChild(document.
getElementById(element));
51:       event.stopPropagation();
52:     }
53:     return false;
54:   };
55: </script>
56: </head>
57:
58: <body>
59:
60:   <div class="cible" id="cible_1" ondragstart="dragStart(event);"
ondragover="return
61:           dragOver(event); ondrop="return drop(event);">
62:     <div id="elt" class="element" draggable="true">
63:       Linux Pratique
64:     </div>
65:   </div>
66:   <div class="cible" id="cible_2" ondragstart="dragStart(event);"
ondragover="return
67:           dragOver(event); ondrop="return drop(event);">
68:     </div>
69:   </body>
70: </html>

```

Vous remarquerez tout d'abord que j'ai inséré les règles CSS et le JavaScript directement dans la page web dans la partie `<head>`, de manière à n'obtenir qu'un seul bloc de code... Il est bien sûr préférable de séparer ces données dans des fichiers distincts.

Ce code me semble suffisamment compréhensible pour se passer de commentaires mis à part, peut-être, pour les lignes 50 et 51. Ces lignes permettent d'ancrez physiquement l'élément déplacé à la cible et de stopper la propagation des événements.

5. Conclusion

Maintenant que la gestion des actions du joueur n'a plus de secret pour vous, vous allez pouvoir commencer à réaliser vos premières ébauches de jeux utilisables. Les articles qui suivent vous permettront de les améliorer, mais nous avons d'ores et déjà vu les bases de la création de jeux en HTML 5. ■

Références

- [1] Table des codes ASCII : <http://www.table-ascii.com>
- [2] Extension Hotkeys pour jQuery : <https://github.com/tzuryby/jquery.hotkeys>



UTILISER DES POLICES DE CARACTÈRES EXTERNES

par Tristan Colombo

Dans le déroulement de votre jeu, il faudra forcément à un moment ou à un autre que vous puissiez « parler » au joueur. Pour cela, la solution la plus simple reste encore d'afficher du texte à l'écran. Mais avez-vous pensé à l'effet que produirait votre texte en **Arial** alors que vous auriez pu l'écrire en **Evil-Black-Dragon-Killer** ?



La problématique des polices de caractères dans un jeu sera sensiblement équivalente à celle que l'on peut rencontrer dans la création d'une page

web ; le texte doit être lisible, la multiplication des polices au sein d'une même page est fortement déconseillée et le choix des polices doit être judicieux : pour évoquer un chuchotement, il vaut mieux éviter une police en *96pt bold* ! L'avantage du jeu par rapport à une page web, c'est que vous allez pouvoir utiliser des polices vraiment exotiques.

Durant l'ère pré-HTML 5, il y a donc de cela fort longtemps, il était impossible d'utiliser une police autre qu'une police installée sur la machine de l'utilisateur. Comme l'on ne peut pas être certain que la police requise ait été effectivement installée, on donne une liste de polices standards, sachant qu'au moins l'une d'entre elles sera présente et pourra donc être utilisée. Le problème de ces polices, c'est qu'elles ne sont pas forcément très élégantes pour une page web... Que dire alors pour un jeu où les graphismes ont un impact important sur l'utilisateur ?

Pour pallier ce défaut de jeunesse, CSS 3 a introduit une nouvelle règle : **@font-face**. Grâce à cette directive « magique », on peut maintenant indiquer au navigateur que l'on souhaite utiliser une police qui devra être téléchargée depuis le serveur. Il faudra s'assurer de bien gérer les différents problèmes pouvant apparaître en chargeant ainsi une police, mais cette action est désormais possible. Nous détaillerons dans cet article quelles sont les contraintes à garder à l'esprit, puis

comment utiliser une police externe grâce à la fameuse règle **@font-face** et également grâce à Google et son service **Google Web Fonts**.

1. Contraintes sur les polices

1.1 Taille des fichiers de polices

Chaque fichier de police peut s'avérer relativement lourd (jusqu'à plusieurs centaines de Ko). De plus, il ne faut pas oublier qu'un fichier correspond à un corps particulier de la police. Ainsi, si vous utilisez une police donnée et que certains mots apparaissent en gras, d'autres en italiques et enfin d'autres en gras et italiques, vous devrez charger un total de quatre fichiers !

Cette contrainte particulièrement importante pour les pages web peut être contournée dans le cas d'un jeu. Lors de la consultation d'un site Internet, un utilisateur n'acceptera pas de patienter sagement qu'une page charge ses polices, alors que si les choses sont bien présentées, cela ne représente aucun problème pour une application et donc, pour un jeu. Il suffit d'ajouter une barre de chargement, ou d'afficher des messages indiquant les étapes de chargement avant le démarrage du jeu. En fonction du thème du jeu, ces étapes peuvent être signalées par des messages humoristiques comme dans *World of Goo*.

Bien sûr, il ne faut pas abuser de la patience de l'utilisateur : si vous décidez

d'utiliser des dizaines de polices différentes (et donc de charger quelques Mo de données), outre un problème d'homogénéité de vos écrans, les utilisateurs penseront vraisemblablement que le jeu ne fonctionne pas et ils ne verront même pas les premières secondes de votre création.

1.2 Licence

Il faut bien entendu vérifier les licences des polices que vous utilisez. Certaines polices seront totalement gratuites, d'autres gratuites seulement pour un usage non commercial et d'autres enfin, nécessiteront l'achat d'une licence. De nombreux sites vous proposeront ces polices et l'on peut citer notamment le site **dafont** [1] sur lequel sont hébergées de très nombreuses polices et qui indique de manière très claire les conditions d'utilisation de chacune d'elles. De plus, il est possible de signaler dans les options de recherche que l'on ne souhaite afficher que les polices du domaine public ou gratuites... Un gain de temps certain pour une recherche qui s'avère toujours extrêmement chronophage ! Le site **FontSquirrel** [2], quant à lui, n'héberge que des polices libres. C'est encore plus simple...

1.3 FOUT

Le FOUT, à ne pas confondre avec un souffle de dépit « PFOUT », est un effet qui peut apparaître sur les navigateurs lors de l'application d'une feuille de styles CSS. FOUT signifie *Flash Of Unstyled Text*, c'est-à-dire flash de texte sans style : le navigateur affiche la page reçue avec les polices système, puis lorsque les polices sont chargées,

il recalcule et ré-affiche la page. Si vous gérez le chargement de vos différents éléments par une barre de progression, ce problème n'apparaîtra pas.

1.4 Les différents formats de polices

Il existe différents formats de fichiers de polices qui permettent tous de stocker quatre types d'informations :

- le tracé vectoriel de l'ensemble des caractères disponibles (certaines polices, par exemple, ne contiennent pas les caractères accentués) ;
- les informations permettant le lissage de la police ;
- les informations de crénage (gestion de l'espace sur certaines juxtapositions de caractères et gestion des ligatures pour les lettres qui peuvent se toucher comme « tt » ou « ff ») ;
- des informations diverses sous forme de métadonnées.

Dans le petit monde du Web, on utilise quatre formats différents :

1.4.1 Polices TrueType et OpenType

Le format TrueType (extension `.ttf`) a été créé par Apple vers la fin des années 1980. Le principal intérêt de ce format a été d'introduire des informations de lissage, permettant d'optimiser simplement le rendu à l'écran.

À la fin des années 1990, Adobe et Microsoft introduisent une évolution du format TrueType nommée OpenType (extension `.otf`). Ce format est une sur-couche à TrueType et exploite également le format « mère » de TrueType, qui est Type 1 PostScript. Il normalise l'encodage des caractères en se basant sur Unicode et permet une gestion très fine des droits d'utilisation : autorisation ou non d'incorporer la police dans un document, etc. C'est du Microsoft... Il s'agit à l'heure actuelle du format de polices le plus utilisé de manière générale, pas pour le Web.

L'inconvénient de ces polices est le poids très élevé des fichiers produits, car s'ils contiennent énormément d'informations de définition et d'optimisation, ils sont forcément volumineux.

1.4.2 Polices Embedded OpenType

À la fin des années 1990, pour résoudre les problèmes posés par les polices OpenType dans un cadre d'utilisation Web, Microsoft propose un format de fichier allégé, compressé, le Embedded OpenType (extension `.eot`). Ce format fermé (on est toujours chez Microsoft...) est exploité par Internet Explorer... et seulement par Internet Explorer. Autant ignorer son existence.

1.4.3 Polices WOFF

Le format WOFF (extension `.woff`) est un format spécialement conçu pour le Web par Mozilla en 2008. Pour simplifier, il s'agit d'un format Embedded OpenType ouvert, en voie de normalisation auprès du W3C et qui devrait donc devenir le format de référence pour le Web.

1.4.4 Polices SVG

Je terminerai cette liste de formats par le format SVG (extension `.svg`). Ce format n'est pas le plus récent, puisqu'il date de la fin des années 1990. C'est un format inventé pour pouvoir embarquer des polices typographiques dans un document SVG : à chaque caractère Unicode on associe un tracé vectoriel. Avantage de ce format : la souplesse de SVG avec l'utilisation des dégradés, filtres, etc. Inconvénient : SVG étant basé sur XML, il est très verbeux et donc les fichiers sont rapidement très volumineux.

1.4.5 Les formats de polices et les navigateurs

Maintenant que nous avons vu ces différents formats, il pourrait être intéressant de savoir quels navigateurs les prennent en charge. Ces informations sont résumées dans le tableau 1 pour les versions suivantes des navigateurs : Firefox 12, Chrome 19, Safari 5.1, Opera 11 et Internet Explorer 10.

On peut constater qu'Internet Explorer ne gère pas les formats ttf/otf (otf étant quand même, je le rappelle, un format créé par Microsoft...). Par contre, il est le seul à pouvoir afficher des polices eot. La prise en charge du SVG est encore rare, avec seulement Safari et Opera. Enfin, le format woff n'est pris en charge que par IE, Firefox et Chrome.

	WOFF	SVG	TTF/OTF	EOT
	Oui	Non	Oui	Non
	Oui	Non	Oui	Non
	Non	Oui	Oui	Non
	Non	Oui	Oui	Non
	Oui	Non	Non	Oui

Tableau 1 : Prise en charge des différents formats de polices par les principaux navigateurs (source : webfonts.info)

Maintenant que nous avons vu les différents formats disponibles, comment les utiliser dans une page web ?

2. La règle CSS 3 @font-face

CSS 3 a introduit la règle `@font-face`, qui permet d'insérer des polices externes. Son utilisation est relativement simple : on indique le fichier de police à charger et on lui donne un nom, puis on l'utilise de manière classique avec une règle `font-family`. Pour tester ce mécanisme, nous allons créer un petit fichier HTML 5 :

```

01: <!DOCTYPE HTML>
02:
03: <html lang="fr">
04:   <head>
05:     <meta charset="utf-8">
06:     <title>Test de polices externes</title>
07:   </head>
08:
09:   <body>
10:     <div>
11:       Ceci est un texte avec la police par défaut.
12:     </div>
13:
14:     <div id="external_font">
15:       Ceci est un texte avec la police externe !
16:     </div>
17:   </body>
18: </html>
```

Nous n'avons bien entendu spécifié aucune règle de style pour l'instant et nos lignes 11 et 15 apparaîtront avec la même police. Ajoutons une ligne permettant de charger une feuille de styles entre les lignes 6 et 7 :

```
06bis: <link rel="stylesheet" href="style.css" />
```

Pour effectuer ce test, j'ai récupéré la police Ballpark (en ttf), sur le site FontSquirrel. Cette police ne dispose que d'un seul corps et nous n'aurons donc qu'un fichier **ballpark_weiner.ttf** auquel nous pourrons faire référence dans notre fichier **style.css** :

```
01: @font-face
02: {
03:   font-family : "BallPark";
04:   src: url("ballpark_weiner.ttf") format("truetype");
05: }
06:
07: #external_font
08: {
09:   font-family: "BallPark", Arial, serif;
10:   font-size: 28pt;
11: }
```

En ligne 3, nous donnons un nom à notre police (ici « BallPark ») et nous utilisons ce nom en ligne 9 pour le texte contenu dans la balise **<div>** ayant pour identifiant **external_font**.

Le résultat obtenu est présenté en figure 1.



Fig. 1 : Utilisation de la règle @font-face pour inclure une police ttf.

Tout fonctionne donc très bien... sauf pour Internet Explorer, qui ne supporte pas le format ttf ! Il va donc falloir ruser, comme dans de nombreux cas avec ce navigateur. Nous allons utiliser l'attribut **local("nom_police")** qui n'est pas reconnu par IE et nous indiquerons au préalable un fichier au format eot qui, lui, ne sera interprété que par IE. Voici la modification du code CSS précédent :

```
01: @font-face
02: {
03:   font-family : "BallPark";
04:   src: url("ballpark_weiner.eot") format("embedded-opentype");
05:   src: local(":"),
06:       url("ballpark_weiner.ttf") format("truetype");
07: }
08:
09: #external_font
10: {
11:   font-family: "BallPark", Arial, serif;
12:   font-size: 28pt;
13: }
```

Nous touchons là du doigt le problème de la prise en charge des polices dans les navigateurs. Heureusement, des solutions alternatives existent ! Le site FontSquirrel propose deux outils :

- **@font-face kits** [3] : vous pourrez trouver à cette adresse des packs complets contenant la police sélectionnée dans les formats woff, ttf, eot, et svg, ainsi que le code HTML et CSS permettant d'utiliser cette police.

Par exemple, en sélectionnant la police Vitamin, puis en cliquant sur le lien « **@font-face Kit** », vous aurez accès à la page de téléchargement présentée en figure 2. Le fichier d'archive au format **.zip** correspondant au kit demandé contient les quatre fichiers de la police Vitamin, les quatre fichiers de la police Vitamin outlined et un fichier **stylesheet.css**, ainsi qu'un fichier de démonstration **demo.html**. Le fichier **stylesheet.css** contient les lignes de code permettant d'utiliser ces polices avec n'importe quel navigateur :

```
01: @font-face {
02:   font-family: 'VitaminRegular';
03:   src: url('VITAMIN_webfont.eot');
04:   src: url('VITAMIN_webfont.eot?#iefix') format('embedded-opentype'),
05:        url('VITAMIN_webfont.woff') format('woff'),
06:        url('VITAMIN_webfont.ttf') format('truetype'),
07:        url('VITAMIN_webfont.svg#VitaminRegular') format('svg');
08:   font-weight: normal;
09:   font-style: normal;
10:
11: }
12:
13: @font-face {
14:   font-family: 'VitaminoutlinedRegular';
15:   src: url('VITAMINO-webfont.eot');
16:   src: url('VITAMINO-webfont.eot?#iefix') format('embedded-opentype'),
17:        url('VITAMINO-webfont.woff') format('woff'),
18:        url('VITAMINO-webfont.ttf') format('truetype'),
19:        url('VITAMINO-webfont.svg#VitaminoutlinedRegular') format('svg');
20:   font-weight: normal;
21:   font-style: normal;
22:
23: }
```

Après ces déclarations, nous pouvons donc utiliser soit la police « **VitaminRegular** », soit la police « **VitaminoutlinedRegular** ». Modifions notre feuille de styles **style.css** en copiant les lignes de ce fichier et en y ajoutant les lignes suivantes :

```
24:
25: #external_font
26: {
27:   font-family: "VitaminRegular", Arial, serif;
28:   font-size: 28pt;
29: }
```

Nous obtenons alors un texte en VitaminRegular (Fig. 3) avec l'assurance de pouvoir exécuter notre code sur n'importe quel navigateur.

- **@font-face generator** [4] : cet outil vous permet de convertir des polices qui ne sont pas forcément proposées par le site FontSquirrel et d'obtenir un paquetage équivalent au **@font-face kit**. La figure 4 montre l'utilisation de cet outil, où la police BallPark a été transmise en cliquant sur le bouton « **Add Fonts** ».

Une autre solution consiste à utiliser un service externe. C'est ce que propose Google.

3. Et Google est arrivé...

Google Web Fonts [5] propose, à l'heure où ces lignes sont écrites, 501 polices. Il suffit de sélectionner sa police et on obtient une page indiquant l'impact sur le temps de chargement de la page (Fig. 5), ainsi que le code HTML à ajouter pour un

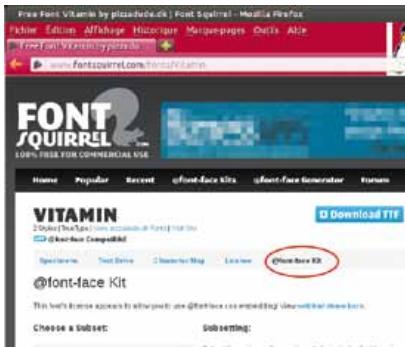


Fig. 2 : Téléchargement du @font-face kit de la police Vitamin sur le site fontsquirrel.com



Fig. 4 : Création d'un @font-face kit de la police BallPark (ttf) depuis l'outil @font-face generator



Fig. 5 : Informations de Google Web Fonts sur une police sélectionnée. Notez l'impact sur le temps de chargement de la page sur la droite.

chargement de la police, éventuellement le code JavaScript si vous préférez utiliser ce type de chargement et le code CSS pour utiliser cette police.

Pour utiliser la police Butcherman, nous devrons écrire le code HTML suivant :

```
<link rel="stylesheet" href="http://fonts.googleapis.com/css?family=Butcherman" />
```

Si vous désirez utiliser plusieurs polices ou des polices italiques, grasses, etc., vous pouvez créer des « collections » de polices, puis cliquer sur le lien « Use » en bas à droite de la page. Si vous connaissez les noms des polices que vous souhaitez utiliser, vous pouvez simplement créer l'URL correspondante :

- l'URL commence par <http://fonts.googleapis.com/css?family=> ;
- les noms des polices suivent et sont séparés par un caractère | et les espaces de leurs noms sont remplacés par le caractère + ;
- le nom de chaque police peut être suivi du caractère : puis d'une lettre indiquant le corps (i ou italic pour italique, b ou bold pour gras, un nombre pour le poids des lettres, et bi pour gras et italique). Pour plusieurs corps d'une même police, séparez les caractères par des virgules.

Ainsi, pour obtenir la police Amaranth en poids 400 et la police Overlock en poids 400 et en poids 700 italique, on va créer l'URL suivante : <http://fonts.googleapis.com/css?family=Amaranth:400|Overlock:400,700italic>

Si vous avez la curiosité de tester cette URL dans un navigateur, vous verrez qu'il ne s'agit ni plus ni moins que d'un fichier de styles CSS contenant des règles @font-face.

Le code JavaScript fourni sur la page du site de Google Web Fonts correspond à ce que Google appelle le « WebFont Loader » : un mécanisme permettant d'éviter les FOUT.



Fig. 3 : Utilisation des fichiers du @font-face kit sur notre exemple

```
01: <script>
02:   WebFontConfig = {
03:     google: { families: [ 'Butcherman::latin' ] }
04:   };
05:   (function() {
06:     var wf = document.createElement('script');
07:     wf.src = ('https:' == document.location.protocol ? 'https' : 'http') +
08:       '//ajax.googleapis.com/ajax/libs/webfont/1/webfont.js';
09:     wf.type = 'text/javascript';
10:     wf.async = 'true';
11:     var s = document.getElementsByTagName('script')[0];
12:     s.parentNode.insertBefore(wf, s);
13:   })(); </script>
```

Attention toutefois à un aspect du service de Google : il ne fournit que des polices en woff, et sans accès au service de Google, pas de police (à moins de télécharger le fichier sur votre serveur)...

4. Conclusion

Nous avons pu voir qu'il était maintenant très simple d'utiliser des polices exotiques dans une page web grâce à la règle @font-face de CSS 3. Si vous ne trouvez pas la police dont vous rêvez sur les différents sites énoncés, il vous reste la possibilité de créer votre propre police à l'aide de **FontForge** [6]. Ce logiciel vous permet de dessiner caractère à caractère votre police, en partant d'une police modèle (ou de rien si vous avez vraiment beaucoup de temps à perdre). C'est une activité fastidieuse et longue, mais qui peut être intéressante pour ajouter une touche vraiment personnelle à vos jeux. ■

Références

- [1] Bibliothèque de polices : <http://www.dafont.com/fr>
- [2] Bibliothèque de polices : <http://www.fontsquirrel.com>
- [3] Outil @font-face kits du site FontSquirrel : <http://www.fontsquirrel.com/fontface>
- [4] Outil @font-face generator du site FontSquirrel : <http://www.fontsquirrel.com/fontface/generator>
- [5] Google Web Fonts : <http://www.google.com/webfonts>
- [6] Site officiel du logiciel de création de polices FontForge : <http://fontforge.sourceforge.net>

GÉRER DES DONNÉES PERSISTANTES

par Tristan Colombo



Milleurs scores, noms de joueurs, paramètres de configuration, ... autant de données qu'il faut conserver en mémoire : inutile de demander au joueur de saisir une cinquantaine de fois son nom !



HTML 5 a introduit la possibilité de stocker des données chez le client. Auparavant, la seule solution disponible était la création de cookies. Le problème de cette approche réside dans le fait que les cookies ne peuvent stocker au maximum que 4Ko de données. Avec les objets de la famille `storage`, nous pourrons stocker jusqu'à 10Mo de données par domaine ! Un autre mécanisme, nommé `indexedDB` permet également de sauvegarder des données chez le client. Ces deux mécanismes ne sont pas incompatibles : le premier permet de stocker des petites quantités de données, alors que le second peut être utilisé pour des données bien plus conséquentes.

Cet article permettra d'appréhender l'utilisation des données persistantes en JavaScript et de tester le mode hors connexion de HTML 5.

1. Les objets `localStorage` et `sessionStorage`

Pour stocker des données, nous utiliserons donc un objet de la famille `storage` (`localStorage`, valable indéfiniment ou `sessionStorage`, valable jusqu'à la fermeture du navigateur). Les données y seront stockées sous forme de tableau associatif : à une clé correspond une valeur. Le stockage d'une valeur se fera alors par l'une ou l'autre des lignes suivantes :

```
localStorage['cle']='valeur';
sessionStorage['cle']='valeur';
```

Pour accéder à une valeur, il faut connaître le nom de la clé (ici par exemple : `alert(sessionStorage['cle']);`), et pour supprimer un élément on utilise `sessionStorage.removeItem('cle')`.

L'extension Firebug pour Firefox peut être très utile, puisqu'elle permet d'afficher le contenu des objets `storage` dans la console de log. Sous Google Chrome, il existe un onglet bien plus pratique dédié à la visualisation des données stockées localement (voir figure 1).

Si vous utilisez ces données pour mettre à jour des champs de formulaire (liste de choix, etc.), n'oubliez pas de rafraîchir l'affichage de vos objets pour que la modification soit prise en compte. Par exemple, en utilisant jQuery sur une liste de sélection nommée `select_ui` et en changeant la valeur de sélection par défaut en fonction de la présence ou non d'une valeur `UI` stockée localement, nous aurons besoin du code suivant :

```
01: if (localStorage['UI']){
02: {
03:   $('#select_ui option[value="'+localStorage['UI']+"]').
04:     attr('selected', 'selected');
05:   $('#select_ui').selectmenu('refresh');
06: }
07: {
08:   localStorage['UI'] = $('#select_ui').val();
09: }
10: );
```

C'est la ligne 4 qui permet de remettre à jour le menu.

Attention : les objets de la famille `storage` sont persistants et synchrones ! Donc si vous les stockez à l'aide de `localStorage` et que le joueur délaisse votre jeu, les données encombreront inutilement la mémoire de son ordinateur. Si vous stockez de grosses données (des images par exemple), la page de votre application sera indisponible le temps du chargement. Cette technique est dédiée au stockage de petites données : des scores, des dates, des noms de joueurs, etc.



Fig. 1 : Barre de développement de Google Chrome permettant la visualisation des données stockées localement

En admettant que l'on souhaite stocker le meilleur score d'un joueur, nous devrons conserver son nom, son score et éventuellement la date. Il faudra donc écrire trois lignes de code (en considérant que la date a déjà été calculée précédemment dans une variable `dateJour`, et que le score se trouve dans la variable `scoreValue`) :

```
localStorage['bestScore_name'] = 'Tristan';
localStorage['bestScore_score'] = scoreValue;
localStorage['bestScore_date'] = dateJour;
```

Ces données ne seront pas facilement manipulables. Il aurait été plus joli de les stocker sous forme d'objets... Mais les objets `storage` ne permettent de stocker que des chaînes de caractères.

2. Stocker des objets

Pour stocker localement des objets, nous allons « sérialiser » nos données : grâce à la fonction `JSON.stringify()` nous allons transformer un objet en une longue chaîne de caractères. Pour « dé-sérialiser » nos données et donc récupérer l'objet de départ, nous utiliserons la fonction `JSON.parse()`. Voici l'adaptation du code précédent en utilisant cette méthode :

```
var bestScore = new Object(
{
    'name': 'Tristan',
    'score': scoreValue,
    'date': dateJour
});
localStorage['bestScore'] = JSON.stringify(bestScore);
```

Lorsque nous voudrons lire les données et les utiliser, nous ferons simplement :

```
var bestScore = JSON.parse(localStorage['bestScore']);
```

3. La base de données indexedDB

L'API `indexedDB` permet une utilisation synchrone ou asynchrone dans la base. Dans la plupart des cas, c'est l'utilisation de l'API asynchrone qui sera bien sûr privilégiée (de toute façon, l'API synchrone n'est pas encore développée...).

La base de données n'est pas relationnelle et n'utilise pas SQL comme langage de requête. Il s'agit d'un modèle objet où

les données ne sont pas stockées sous forme de tables et de collections mais sous forme de couples clé/valeur, comme pour les objets de la famille `storage`. Les étapes nécessaires à l'utilisation de `indexedDB` sont les suivantes :

- Récupération de l'objet `indexedDB` en fonction du type de navigateur ;
- Ouverture de la base de données ;
- Création d'un `objectStore` pour stocker des données ou bien récupération d'un `objectStore` pour lire des données. L'`objectStore` est en quelque sorte le pendant de la notion de table : une manière d'identifier une collection de données. Toutefois, le « schéma » défini par cet objet ne sera pas statique et vous pourrez ajouter à tout moment de nouveaux champs ;
- Utilisation des données.

Nous en resterons à la théorie pour cette base : la norme du W3C n'est pas figée pour HTML 5 et elle est même très changeante concernant cette API. De plus, l'implémentation au niveau des différents navigateurs est assez exotique et rien n'est vraiment satisfaisant. Pour en savoir un peu plus, je vous renvoie vers un article du *Mozilla Developer Network* [1] et une présentation de chez Google [2] (voir le code du slide 34 qui commence en ligne 1090). Vous comprendrez alors qu'il est préférable d'attendre que la norme soit stabilisée avant d'utiliser cette technologie.

4. Le mode hors connexion

Pour que les joueurs puissent utiliser votre jeu même sans connexion, vous pouvez utiliser le mode hors connexion de HTML 5. Cette mise en cache rend les pages statiques disponibles hors connexion, mais attention à son utilisation :

- La mise en cache n'est pas automatique, l'utilisateur doit l'accepter ;
- Le cache est prioritaire sur la version en ligne !

Pour activer le cache, il faudra créer un fichier d'extension `manifest` qui sera chargé depuis la balise `<html>` de votre document principal (ici, le fichier s'appelle `offline.manifest`) :

```
<html manifest="offline.manifest">
```

Le fichier `manifest` se compose de trois parties :

- `CACHE` : liste des fichiers à mettre en cache ;
- `NETWORK` : liste des fichiers à ne pas mettre en cache ;
- `FALLBACK` : fichier à exécuter si une ressource est indisponible.

La syntaxe exacte du fichier est la suivante :

CACHE MANIFEST

```
CACHE:
index.html
image.png
```

```
NETWORK:
connect.php
```

```
FALLBACK:
/repertoire/offline.html
```

Pour savoir si l'utilisateur est en ligne ou non, vous pourrez utiliser la propriété `navigator.onLine`, qui renvoie un booléen (`true` si le navigateur est en mode connecté et `false` si le navigateur est en mode hors connexion). De plus, la classe `applicationCache` permet de mettre à jour le cache :

```
var webappCache = window.applicationCache;
webappCache.update();
```

5. Conclusion

Toutes les données d'un jeu ne doivent pas forcément être rechargées ou demandées à l'utilisateur à chaque lancement. Nous avons vu quelques méthodes permettant de conserver des données en mémoire en partant de données de petites tailles avec `localStorage` et `sessionStorage`, jusqu'aux données de grande taille avec `indexedDB` et la mise en cache.

Il est dommage que `indexedDB` ne puisse pas vraiment être utilisé sans risque (à moins d'utiliser une API encore « bancale » et de se restreindre à un navigateur et à une version spécifique de celui-ci). Mais ne soyons pas trop exigeants, nous avons déjà beaucoup plus d'outils que sous HTML 4... ■

Références

[1] Tutoriel d'utilisation d'indexedDB : https://developer.mozilla.org/en/IndexedDB/Using_IndexedDB

[2] Présentation sur indexedDB : <http://html5-demos.appspot.com/static/html5storage/index.html#slide34>



LE MODE MULTIJOUEUR

par Tristan Colombo

Pour que plusieurs joueurs puissent se retrouver immergés au sein du même environnement, vous n'aurez d'autre choix que de passer par un serveur. C'est lui qui centralisera les données de chaque joueur et permettra d'introduire un peu d'intelligence pour la gestion des personnages non-joueurs.



Pour communiquer avec un serveur, le protocole HTTP n'est pas ce qui se fait de mieux. En effet, il n'est pas prévu pour des interactions en temps réel et, à

chaque requête, il faut ouvrir une nouvelle connexion. Pour contourner le problème, différentes stratégies ont été mises en place comme les méthodes Ajax et Comet. Mais il ne s'agit que d'alternatives que l'on peut qualifier d'« instables », car ne reposant sur aucune véritable norme et utilisant l'objet **XMLHttpRequest** de manière dérivée.

La norme HTML 5 a vu arriver un protocole de communication bidirectionnel et *full duplex* sur un socket TCP : les websockets. Ce protocole standardisé permet d'envoyer et de recevoir des données en continu (comme avec Comet, mais en plus propre). Dans cet article, nous aborderons la théorie et l'utilisation de ce protocole qui permettra de gérer l'intelligence artificielle du jeu et de multiples joueurs.

1. Les websockets en théorie

Il faut savoir que les websockets représentent un protocole en cours de développement dont l'API n'est pas forcément stable. Ainsi, sur les dernières versions de Firefox est-on passé par plusieurs noms pour représenter le même objet, une désactivation automatique du protocole, etc. Pour savoir si votre navigateur supporte les websockets, rendez-vous sur la page

<http://www.websocket.org> et regardez le cadre en haut à droite de la page. Vous devriez y voir une image semblable à celle de la figure 1 (en tout cas si vous utilisez Firefox 14... sous Linux, bien sûr). Normalement, avec les dernières versions des différents navigateurs, vous devriez avoir en standard un support des websockets.



Fig. 1 : Vérification du support des websockets sur un navigateur à partir du site <http://www.websocket.org>

La connexion à un serveur de websockets se fait en JavaScript par une URL de la forme **ws://** ou **wss://** pour la version sécurisée (voir <http://websocket.org/echo.html> pour une démonstration). Cette connexion passe par la création d'une instance de l'objet **WebSocket** :

```
var ws= new WebSocket('ws://websocket_url');
```

Quatre événements peuvent être associés à cet objet :

- **onopen** : ouverture d'une websocket ;
- **onmessage** : réception d'un message ;
- **onerror** : erreur(s) survenue(s) ;
- **onclose** : fermeture d'une websocket ;

Ensuite, pour envoyer des messages ou des données, on utilise la fonction **send()** et la fonction **close()** pour fermer la connexion. En utilisant l'objet **WebSocket** défini précédemment, nous pourrions ainsi avoir :

```
ws.onopen = function (event) { alert('Ouverture de la connexion...'); };
ws.onmessage = function (event) { alert('Message reçu : ' + event.data); };
ws.onclose = function (event) { alert('Fermeture de la connexion'); };
ws.onerror = function (event) { alert('Erreur : ' + event.data) };

myWebSocket.send('Ceci est un message !');

myWebSocket.close();
```

La connexion au serveur de websockets se gère donc très simplement... Mais qui dit connexion à un serveur, dit justement serveur... Comment écrire un serveur de websockets ? De nombreux serveurs sont en cours de développement : je ne citerai que ceux qui seront utilisés dans le cadre de cet article, à savoir AutobahnPython [1], qui comme son nom l'indique, est en Python et Socket.IO [2] en JavaScript, basé sur node.js [3].

2. Les websockets en pratique

Nous allons implémenter deux serveurs de websockets à partir d'un même exemple : nous allons simuler le comportement d'un ou plusieurs client(s) désirant commander des bières à un serveur dans un bar. La première implémentation sera réalisée avec AutobahnPython qui gère les websockets tel que défini par le W3C. La seconde implémentation sera réalisée avec Socket.IO, qui lui aussi gère les websockets, mais réalise une encapsulation lui permettant d'utiliser d'autres techniques si les websockets ne sont pas supportées par le navigateur. Un autre intérêt de Socket.IO pour le développement Web est de ne pas ajouter un énième langage dans la pile des langages déjà utilisés. Vous savez coder en JavaScript ? Eh bien ce sera largement suffisant.

2.1 AutobahnPython

AutobahnPython est un framework Python dédié à l'utilisation des websockets. Il est basé sur le module de communication réseau Twisted.

2.1.1 Installation

L'installation se fait de manière totalement automatisée depuis PyPi (*PYthon Package Index*) grâce à la commande **pip** qui gérera les dépendances (avec par exemple l'installation de Twisted) :

```
sudo pip install autobahn
```

En cas d'erreur sur la compilation de Twisted, c'est qu'il vous manque vraisemblablement le paquetage **python-dev** :

```
sudo aptitude install python-dev
```

2.1.2 Le serveur

Ni Autobahn, ni Twisted n'ayant encore été portés en Python 3, c'est en Python 2 que nous écrirons notre code. Voici le serveur contenu dans le fichier **server.js** :

```

01: # -*- coding:utf-8 -*-
02:
03: import sys
04:
05: from twisted.internet import reactor
06: from twisted.python import log
07:
08: from autobahn.websocket import WebSocketServerFactory, \
09:                             WebSocketServerProtocol, \
10:                             listenWS
11:
12: import json
13: import uuid
14:
15:
16: class ServerProtocol(WebSocketServerProtocol):
17:
18:     def onOpen(self):
19:         """
20:             Actions à effectuer à la connexion
21:
22:             print '>>> Connexion effectuée'
23:             self.uid = str(uuid.uuid1())
24:             data = { 'type': 'Connected', 'msg': 'Bonjour, je suis le serveur!' }
25:             self.sendMessage(json.dumps(data), False)
26:
27:     def onMessage(self, data, binary):
28:         d = json.loads(data)
29:
30:         if d['type'] == 'Message':
31:             """
32:                 Réception des messages
33:             """
34:             print '>>> Message reçu : ' + d['msg'].encode('ascii')
35:
36:         elif d['type'] == 'Commande':
37:             """
38:                 Réception des commandes
39:             """
40:             print '>>> Commande reçue ' + str(d['nb']) + ' ' + \
41:                  d['boisson'].encode('utf-8') + '(s) pour le client ' + \
42:                  str(self.uid)
43:             response = { 'type': 'Serveur', 'msg': 'Et ' + str(d['nb']) + ' ' + \
44:                         d['boisson'].encode('utf-8') + '(s) pour la table ' + \
45:                         str(self.uid) }
46:             self.sendMessage(json.dumps(response), False)
47:
48:     def onClose(self, clean, code, msg):
49:         """
50:             Actions à effectuer à la déconnexion
51:
52:             print '>>> Le client ' + str(self.uid) + ' est parti :-('
53:
54: if __name__ == '__main__':
55:     Log.startLogging(sys.stdout)
56:
57:     factory = WebSocketServerFactory('ws://localhost:3000', debug = False)
58:     factory.protocol = ServerProtocol
59:     listenWS(factory)
60:
61:     reactor.run()

```

Les imports des lignes 3 à 10 sont nécessaires pour utiliser Autobahn. Les imports des lignes 12 et 13 permettront de manipuler le format JSON et d'obtenir un identifiant unique. Nous définissons ensuite une classe **ServerProtocol** qui hérite de la classe **WebSocketServerProtocol** (lignes 16 à 52). Cette classe va nous permettre de gérer toutes les actions à effectuer en fonction des messages reçus.

Lors de la connexion, la méthode **onOpen()** des lignes 18 à 25 est ainsi appelée. Elle affiche alors un message de débogage sur le serveur indiquant que la connexion a été

effectuée (ligne 22), elle crée un nouvel attribut `uid` contenant un identifiant unique permettant d'identifier la connexion (ligne 23), et elle renvoie un message contenant des données au format JSON (ligne 24), qui ont été serialisées par la fonction `json.dumps()`.

C'est la méthode `sendMessage()` de la ligne 25 qui envoie l'information donnée en premier paramètre, le second paramètre indiquant que les données ne sont pas binaires. En cas de réception d'un message, ce sera la méthode `onMessage()` qui sera appelée (lignes 27 à 46). Comme les données récupérées dans la variable `data` sont serialisées, il faut les dé-sérialiser à l'aide de la fonction `json.loads()` de la ligne 28. Ensuite, en fonction de la valeur du champ `d['type']`, diverses actions sont effectuées : soit il s'agit d'un simple message (ligne 30) et un message sera affiché sur le serveur (ligne 34), soit il s'agit d'une commande (ligne 36) et un message sera affiché (lignes 40 à 42) pendant qu'un autre sera envoyé au client (lignes 43 à 46). Notez que pour gérer ces différents cas vous pouvez aussi créer un aiguillage qui associera à chaque label un pointeur vers une fonction. En Python, cela se fait très simplement :

```
dispatcher = {
    'Message': message,
    ...
}
```

Il suffit ainsi d'appeler ensuite `dispatcher(d['type'])(d)` pour lancer les fonctions.

Pour revenir à notre code, la méthode `onClose()` des lignes 48 à 52 est exécutée à la perte de la connexion avec le client. Le paramètre `clean` est un booléen indiquant si la connexion a été fermée « proprement » et si ce n'est pas le cas, `code` et `msg` contiendront le code et le message d'erreur. Lors de la déconnexion, nous affichons un message indiquant l'identifiant du client qui s'est déconnecté.

Enfin, le programme principal se trouve dans les lignes 54 à 61. En ligne 55, nous demandons l'affichage des informations de log à l'écran, puis nous créons la connexion dans les lignes suivantes : le protocole est `ws` en local sur le port 3000 (ligne 57). Avant d'utiliser le client, il faudra penser à lancer le serveur par :

```
python server.py
```

2.1.3 Le client

Les fichiers HTML et CSS sont très simples :

```
01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Websockets avec socket.io</title>
06:     <script src="client.js"></script>
07:     <link rel="stylesheet" href="style.css" media="screen"></style>
08:   </head>
09:
10:   <body>
11:     <div class="box">Serveur : <span id="serveur">ZZZZzzzzZZZZ</span></div>
12:     <a href="#" class="button" onclick="msg('Bonjour, je suis le client')">
```

```
13:       Saluer le serveur
14:     </a><br />
15:     <a href="#" class="button" onclick="cmd(1, 'bière')">
16:       Commander une pression
17:     </a><br />
18:     <a href="#" class="button" onclick="bye()">
19:       Dire au revoir
20:     </a><br />
21:   </body>
22: </html>
```

Notre page affichera un champ de notification (ligne 11) et possédera trois boutons :

- un bouton pour saluer le serveur (lignes 12 à 14),
- un bouton pour commander une bière (lignes 15 à 17),
- et un bouton pour partir et donc couper la communication (lignes 18 à 20).

Chacun de ces boutons fait appel à une fonction définie dans le fichier JavaScript `client.js` chargé en ligne 6.

Le code CSS se passe de commentaires et n'est donné qu'à titre indicatif :

```
01: .box
02: {
03:   width: 400px;
04:   margin: auto;
05:   margin-bottom: 50px;
06:   padding: 2px;
07:   border: 1px solid #000;
08:   background-color: #b1a7a7;
09: }
10:
11: .button
12: {
13:   display: block;
14:   width: 300px;
15:   margin-bottom: 10px;
16:   border: 1px solid #000;
17:   background-color: #c7b219;
18: }
```

Voici le code du fichier `client.js` effectuant la connexion et le dialogue avec le serveur :

```
01: var ws = new WebSocket('ws://localhost:3000');
02:
03: function display_serveur(message)
04: {
05:   console.log(message);
06:   document.getElementById("serveur").innerHTML = message;
07: }
08:
09: function msg(texte)
10: {
11:   var data = { type: 'Message', msg: texte };
12:   ws.send(JSON.stringify(data));
13: }
14:
15: function cmd(n, b)
16: {
17:   cmd.cpt+=n;
18:   var data = { type: 'Commande', nb: cmd.cpt, boisson: b };
19:   ws.send(JSON.stringify(data));
20: }
21:
```

```

22: function bye()
23: {
24:     ws.close();
25: };
26:
27: window.onload = function ()
28: {
29:     cmd.cpt = 0;
30:
31:     ws.onmessage = function(event)
32:     {
33:         var data = JSON.parse(event.data);
34:
35:         if ((data.type == 'Connected') || (data.type == 'Serveur'))
36:         {
37:             display_serveur(data.msg);
38:         }
39:     }
40: };

```

La ligne 1 crée la connexion avec le serveur sur le port 3000 grâce à l'objet `WebSocket`. La fonction `display_serveur()` des lignes 3 à 7 affiche le message transmis en paramètre dans la console de Firebug (ou de Chrome) et met à jour la zone de notification identifiée par `serveur` (ligne 6).

La fonction `message()` des lignes 9 à 13 envoie un message au serveur. Il s'agit d'un objet JSON (ligne 11) qui est sérialisé par `JSON.stringify()` avant envoi par la méthode `send()` (ligne 12). La fonction `cmd()` des lignes 15 à 20 réalise la même opération tout en maintenant à jour la valeur d'une variable statique `cmd.cpt` contenant le nombre total de boissons commandées (ligne 17).

La fonction `bye()` des lignes 22 à 25 permet de se déconnecter du serveur grâce à la méthode `close()` (ligne 24). Enfin, au chargement de la page, nous initialisons le nombre de commandes à 0 (ligne 29) et lorsqu'un message est envoyé par le serveur (événement `message` sur l'objet `WebSocket`, comme le montre la ligne 31), nous récupérons les données et les affichons (lignes 31 à 39).

Cette méthode permet d'utiliser la véritable API websockets de HTML 5. Son point faible est la nécessité de connaître Python pour écrire le serveur. Bien entendu, si vous maîtrisez ce langage, c'est toujours un plaisir de coder en Python... Dans le cas contraire, il vaudra mieux que vous vous penchiez sur l'alternative de Socket.IO.

2.2 Socket.IO

Socket.IO est un module de Node.js, un outil permettant d'exécuter des applications JavaScript en ligne de commandes (serveur en JavaScript, gestion des entrées/sorties de manière non bloquante, etc.) en utilisant le moteur V8 de Google Chrome (Chromium pour sa version Linux). Node.js n'utilise qu'un seul thread pour gérer les requêtes entrantes et ne propose pas dans son API de fonctions bloquantes. Tout le code est asynchrone ce qui simplifie énormément la gestion de la concurrence, puisqu'elle est déléguée au système d'exploitation. Node.js va utiliser une boucle d'événements au lieu de threads. Il s'agit donc d'un système événementiel.

En pratique, toutes les fonctions fournies par Node.js prennent en paramètre une fonction de rappel. Quand la fonction

Node.js aura achevé son traitement, la fonction de rappel sera appelée avec en paramètres le résultat de la fonction mère et une éventuelle erreur.

2.2.1 Installation

L'installation de Node.js et Socket.IO se fait très simplement en utilisant le gestionnaire de paquetages de votre distribution. Pour une distribution basée sur Debian, il faudra faire :

```

sudo aptitude install build-essential nodejs npm
sudo npm install socket.io

```

Notez que la dernière commande crée un sous-répertoire `node_modules` dans le répertoire dans lequel elle a été exécutée. Il s'agit d'une installation locale du module Socket.IO. Si vous souhaitez réaliser une installation globale, ajoutez l'option `-g` :

```

sudo npm install -g socket.io

```

Si vous le souhaitez, vous pouvez également installer un débogueur en mode graphique : `node-inspector`.

```

sudo npm install -g node-inspector

```

Pour l'utiliser, il faudra bien sûr le lancer :

```

node-inspector &

```

Puis exécuter votre code Node.js avec l'option `--debug` :

```

node --debug monServer.py

```

Et enfin, ouvrir votre navigateur à l'adresse : <http://127.0.0.1:8080/debug?port=5858>.

2.2.2 Le serveur

Comme avec Autobahn, nous allons commencer par écrire le serveur en JavaScript. Le fichier se nommera `server.js` et déterminera le comportement à adopter pour envoyer et recevoir des messages.

```

01: var io = require('socket.io').listen(3000);
02:
03: io.sockets.on('connection', function (socket)
04: {
05:     // Actions à effectuer à la connexion
06:     console.log('>>> Connexion effectuée');
07:     socket.emit('Connected', { data : 'Bonjour, je suis le serveur !' });
08:
09:     // Réception des messages
10:     socket.on('Message', function (msg)
11:     {
12:         console.log('>>> Message reçu : ' + msg.data);
13:     });
14: };
15:
16: // Réception des commandes
17: socket.on('Commande', function (msg)
18: {
19:     console.log('>>> Commande reçue : ' + msg.nb + ' ' + msg.boisson +
20:                 ' pour le client ' + socket.id);
21:     socket.emit('Serveur', { data : 'Et ' + msg.nb + ' ' + msg.boisson +
22:                             ' pour la table ' + socket.id + ' !' });
23: });
24: );

```

```

25:     socket.on('disconnect', function ()
26:     {
27:         console.log('>>> Le client ' + socket.id + ' est parti :-(');
28:     }
29: );
30: );
31: );
32: );

```

Le module `socket.io` est recherché automatiquement dans le répertoire `node_modules` si vous l'avez installé localement. Il n'est plus nécessaire d'ajouter la ligne indiquant où trouver le module comme cela était nécessaire dans les anciennes versions de Node.js :

```
require.paths.unshift(__dirname + '/node_modules/socket.io/lib/');
```

Il est même recommandé de ne pas le faire, puisque votre programme ne fonctionnerait pas, l'instruction étant obsolète... Pour revenir au programme, en ligne 1, nous créons un objet `Socket.IO` en demandant d'écouter sur le port 3000. La suite du code ne contient que des descriptions d'événements.

Lors de la connexion d'un client (ligne 3), nous définissons la liste des événements qui vont être déclenchés sur le socket (lignes 4 à 32). La première action, qui elle, n'est pas conditionnée par un événement autre que la connexion, est l'affichage d'un message sur le terminal du serveur en ligne 6 et l'émission d'un message vers le client dont l'étiquette sera « `Connected` » (ligne 7). Attention : la commande `console.log()` ne fait pas du tout référence à Firebug ici : c'est la commande qui permet d'afficher un message sur le terminal qui a lancé Node.js (on retrouve d'ailleurs la même gamme de fonctions que dans Firebug, avec `console.warn()`, `console.error()`, etc.).

Lors de la réception de données ayant pour étiquette « `Message` » (ligne 10), nous affichons un message sur le terminal indiquant la nature du message (ligne 12). De la même manière, lors de la réception de données étiquetées « `Commande` », nous affichons un message de contrôle sur le terminal en indiquant l'identifiant du client par `socket.id` (lignes 19 et 20), puis nous émettons un message portant l'étiquette « `Serveur` » à destination du client (lignes 21 et 22). Enfin, les lignes 26 à 30 gèrent l'événement de déconnexion d'un client en affichant seulement un message sur le terminal.

2.2.3 Le client

La partie cliente CSS est exactement la même que précédemment et la partie HTML ne contient qu'une modification : l'ajout du chargement du fichier `socket.io.js` après la ligne 5 :

```
05bis: <script src="http://localhost:3000/socket.io/socket.io.js"></script>
```

Pour la partie JavaScript, il s'agit du fichier `client.js` :

```

01: var socket = io.connect('http://localhost:3000');
02:
03: socket.on('Connected', function (msg)
04: {
05:     display_serveur(msg.data);
06: }
07: );
08:

```

```

09: socket.on('Serveur', function (msg)
10: {
11:     display_serveur(msg.data);
12: }
13: );
14:
15: function display_serveur(message)
16: {
17:     console.log(message);
18:     document.getElementById("serveur").innerHTML = message;
19: }
20:
21: function msg(texte)
22: {
23:     socket.emit("Message", { data : texte });
24: }
25:
26: function cmd(n, b)
27: {
28:     cmd.cpt++;
29:     socket.emit("Commande", { nb : cmd.cpt, boisson : b });
30: }
31:
32: function bye()
33: {
34:     socket.disconnect();
35: }
36:
37: window.onload = function ()
38: {
39:     cmd.cpt = 0;
40: }

```

Ce code est relativement simple, le fonctionnement étant très semblable à celui du serveur : nous nous connectons au serveur en ligne 1, puis nous définissons les différents événements à intercepter.

À la réception de données étiquetées « `Connected` » (ligne 3) ou « `Serveur` » (ligne 9), nous appelons la fonction `display_serveur()` des lignes 15 à 19 (même comportement qu'avec Autobahn). Les fonctions `msg()` des lignes 21 à 24 et `cmd()` des lignes 26 à 30 émettent des données étiquetées vers le serveur. Les données sont au format JSON. La dernière fonction `bye()` des lignes 32 à 35 permet de se déconnecter. Enfin, au chargement de la page, nous initialisons la variable statique `cmd.cpt` (ligne 39).

3. Conclusion

Le protocole websocket n'est pas encore stabilisé, mais est très prometteur. Nous pouvons commencer à utiliser l'API définie par le W3C grâce aux éditeurs de navigateurs prenant en compte ces commandes et à des frameworks permettant d'écrire des serveurs respectant ces normes. Si vous voulez vous assurer une compatibilité avec d'anciens navigateurs, `Socket.IO` vous permettra d'écrire un serveur JavaScript performant. Cette technologie vous ouvre donc les portes du mode multijoueur ! ■

Références

- [1] Site officiel de AutobahnPython : <http://autobahn.ws/>
- [2] Site officiel de Socket.IO : <http://socket.io/>
- [3] Site officiel de Node.js : <http://nodejs.org/>

Complétez votre collection de



au tarif promotionnel de 2€^{TTC} par numéro* !

Les 4 façons de commander !

Par courrier

En nous renvoyant ce bon de commande.

Par le Web

Sur notre site : www.ed-diamond.com.

Par téléphone

Entre 9h-12h & 14h-18h au 03 67 10 00 20
(palemban C.B.)

Par fax

Au 03 67 10 00 21
(C.B. et/ou bon de commande administratif)

Choisissez vos numéros dans le tableau ci-dessous : **Linux Pratique**

<input type="checkbox"/> N°21	KPovModeler
<input type="checkbox"/> N°22	eDonkey, eMule, KaZaA - Partagez vos fichiers
<input type="checkbox"/> N°27	Maîtrisez Firefox
<input type="checkbox"/> N°28	Chattez librement !
<input type="checkbox"/> N°29	Créez votre WEBLOG
<input type="checkbox"/> N°30	Quoi de neuf côté ... VIDÉO ?
<input type="checkbox"/> N°31	Tout sur Gnome 2.12
<input type="checkbox"/> N°32	OpenOffice 2.0
<input type="checkbox"/> N°33	Firefox 1.5 - Le dernier né de la mozilla corporation
<input type="checkbox"/> N°34	Les nouveautés de Thunderbird 1.5 - Un concurrent sérieux pour Outlook
<input type="checkbox"/> N°35	Quel avenir pour les bureaux Linux
<input type="checkbox"/> N°36	ClamAV : Configurez simplement votre anti-virus
<input type="checkbox"/> N°37	XFCE 4.4 - Votre prochain bureau Linux ?
<input checked="" type="checkbox"/> N°38	Mandriva passe à la 3D
<input type="checkbox"/> N°39	Les nouvelles fonctionnalités de Firefox 2.0 - Le meilleur navigateur web ?
<input type="checkbox"/> N°40	Knoppix 5.11 - Enfin une solution de lecture/écriture sur disques Xp et Vista
<input type="checkbox"/> N°41	Téléchargement peer to peer facile avec BitTorrent & KDE
<input type="checkbox"/> N°42	FEDORA 7 - Enfin un bureau 3D à la hauteur de vos attentes
<input type="checkbox"/> N°43	KDE, GNOME, ENLIGHTEMENT - Quel sera votre prochain environnement de bureau ?
<input type="checkbox"/> N°44	Firefox 3.0
<input type="checkbox"/> N°45	KDE 4.0

Linux Pratique Hors-Série

Hors-Série N°02	Testez des Logiciels Libres sans rien installer !
Hors-Série N°03	KDE 3.5
Hors-Série N°06	Changez de système d'exploitation : Découvrez Ubuntu Dapper Drake 6.06
Hors-Série N°07	Le dessin vectoriel Libre par la pratique !
Hors-Série N°08	GNU/Linux et les Logiciels Libres en 80 questions - Vol.1
Hors-Série N°09	Découvrez les principales fonctionnalités d'OpenOffice.org 2.0

dans la limite des stocks disponibles.

Bon de commande

à remplir (ou photocopier) et à retourner aux Éditions Diamond - GNU/Linux Magazine - BP 20142 - 67603 Sélestat Cedex

Quantité	Prix / N°	Total
	x 2,00 €	=
FRAIS DE PORT FRANCE MÉTRO. :		+ 3,90 €
FRAIS DE PORT HORS FRANCE MÉTRO. :		NOUS CONSULTER
TOTAL :		

Je choisis de régler par :

Chèque bancaire ou postal à l'ordre des Éditions Diamond

Carte bancaire n°

Expiré le :

Cryptogramme visuel :

Date et signature obligatoire



Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Téléphone :

e-mail :

- Je souhaite recevoir des infos des Éditions Diamond
 Je souhaite recevoir des infos des partenaires des Éditions Diamond



UN PEU DE MUSIQUE

dans ce monde de brutes

par Tristan Colombo

Dans un jeu, la musique est aussi importante que les graphismes d'un point de vue subjectif. Qui ne se souvient pas de la musique de Zelda après quelques notes ? Qui ne connaît pas par cœur les répliques des unités de Warcraft II : « Yes, my lord ! », « Ready to serve », « Cap'tain on the bridge », « ZomZog ! », etc. ? Les effets sonores et la musique de fond marquent le joueur. Ils font partie intégrante du jeu et ne doivent surtout pas être négligés.



La partie la plus longue dans la mise en place des effets sonores d'un jeu est la recherche du son elle-même. Il y a très peu de ressources sur Internet et là où une navigation rapide suffit avec les graphismes (d'un coup d'œil vous savez si l'une des 20 icônes présentes sur la page pourra vous intéresser), vous devrez passer plus de temps avec les sons, puisqu'il faudra les lancer et les écouter. Bien sûr, rien ne remplacera l'artiste créant les sons et musiques sur demande, mais pour commencer, on peut utiliser des sons mis à disposition sur le Web. Parmi les sites « utilisables », voici quelques liens :

- <http://www.bfxr.net/> : outil en ligne permettant de créer des sons 8-bits de manière aléatoire en sélectionnant un thème (explosion, laser, etc.). Il est possible de générer manuellement un son en augmentant ou en diminuant l'intensité des différents effets. Le son créé pourra être exporté au format WAV.
- <http://www.grsites.com/archive/sounds/> : bibliothèque d'effets sonores au format WAV ou MP3. Les sons sont organisés en catégories : nature, voitures, etc.
- <http://soundbible.com/> : là encore, il s'agit d'une bibliothèque d'effets sonores au format WAV ou MP3. Un moteur de recherche permet d'accélérer, ou au moins de rendre un peu plus rapide, la découverte de l'effet convoité.

- <http://www.findsounds.com/> : recherche d'effets sonores uniquement par mots-clés. On peut sélectionner le format attendu (WAV ou MP3).

- <http://www.newgrounds.com/audio/> : vous pourrez trouver sur ce site des musiques d'ambiance au format MP3.

Une fois que vous aurez réussi tant bien que mal à créer votre collection d'effets sonores et de musiques, il va falloir décider à quel moment les jouer. Une ambiance joyeuse et sautillante pour traverser de nuit un cimetière où grouillent goules et morts-vivants ne serait pas forcément la plus appropriée...

De plus les musiques d'ambiance (ou de fond) sont exécutées en boucle, de manière à ne pas être limitées dans le temps. Veillez à ce que votre musique ne soit pas trop répétitive pour ne pas faire fuir le pauvre joueur... Une bonne musique contribue à un bon jeu, mais une musique horripilante, même si le *gameplay* est formidable, sera totalement rédhibitoire pour le jeu. Il en va de même avec les effets sonores. Je vais reprendre pour cela l'exemple de Warcraft II : lorsque l'on cliquait plusieurs fois sur une même unité, on n'obtenait pas indéfiniment la même phrase (ce qui aurait été lassant...). Même sur un effet sonore anodin, comme un tir de laser ou une explosion, il est intéressant de posséder au moins trois ou

quatre sons différents pour établir un roulement et que l'utilisateur ne soit pas exaspéré par des répétitions musicales.

Ces considérations préliminaires étant faites, je vous propose dans cet article de voir comment utiliser la balise `<audio>` introduite par HTML 5 pour pouvoir jouer un son. Nous aborderons ensuite le contrôle de ces sons depuis JavaScript et pour conclure, nous étudierons un aspect un peu plus technique avec la création et l'utilisation de sprites audio.

1. La balise `<audio>`

La balise `<audio>` permet de lire différents formats de fichiers :

- les fichiers Ogg (extension `.ogg`) : données compressées à l'aide du codec Vorbis. Ce format est libre ! Donc à choisir préférentiellement ? Certes, mais bien entendu, Internet Explorer et Safari ne supportent pas ce format...
- les fichiers MP3 (*MPEG-1/2 Audio Layer 3* de son nom complet, extension `.mp3`) : format de compression audio très utilisé de par sa capacité à compresser les fichiers de musique tout en conservant une qualité acceptable et la possibilité de stocker des métadonnées.
- les fichiers AAC (pour *Advanced Audio Coding* soit Encodage Audio

Avancé, extension **.aac**) : format de compression audio avec un meilleur rapport qualité/débit binaire. Ce format est supporté par Safari et Google Chrome.

- les fichiers Wave (extension **.wav**) : format beaucoup plus lourd que tous les précédents. À éviter pour réduire les temps de chargement !

Comme tous ces formats ne sont bien sûr pas pris en charge par tous les navigateurs, le tableau 1 résume le support des différents formats présentés (Firefox 12, Chrome 19, Safari 5.1, Opera 11 et Internet Explorer 10).

	Ogg	MP3	AAC	WAV
	Oui	Non	Non	Oui
	Oui	Oui	Oui	Oui
	Non	Oui	Oui	Oui
	Oui	Non	Non	Oui
	Non	Oui	Non	Non

Tableau 1 : Prise en charge des différents formats audio par les principaux navigateurs

Comme tous les navigateurs ne sont pas capables de lire un même format de fichier audio, il va falloir fournir plusieurs fichiers... Pour convertir simplement les fichiers audio vous pouvez utiliser le logiciel **SoundConverter** [1] ou le logiciel, un peu plus complexe mais plus complet, **Audacity** [2]. Tous deux sont disponibles dans les dépôts des distributions basées sur Debian.

Une fois que vous disposez de vos fichiers audio dans tous les formats désirés, il reste à les insérer dans vos pages HTML. Nous allons donc utiliser la balise **<audio>** et des balises **<source>** pour indiquer les différents fichiers à télécharger (un par format). Quand un format n'est pas pris en charge par le navigateur, la ligne est ignorée et on passe à la suivante :

```

01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test de la balise audio</title>
06:   </head>
07:
08:   <body>
09:     <audio controls>
10:       <source src="bell.mp3" type="audio/mp3" />
11:       <source src="bell.ogg" type="audio/ogg" />
12:       <source src="bell.aac" type="audio/aac" />
13:       Votre navigateur ne supporte pas la balise audio!
14:     </audio>
15:   </body>
16: </html>

```

Dans cet exemple, nous avons utilisé en ligne 9 un attribut **controls** pour la balise **<audio>**. Cet attribut permet d'afficher une barre de contrôle simpliste avec un bouton **start/pause**, une ligne de durée et un bouton **mute** (Fig. 1). Dans la définition formelle issue du W3C [3], cet attribut est booléen et sa seule présence suffit à l'activer. Pour des raisons de notation XHTML, vous pourrez aussi rencontrer **controls="controls"**, mais la première écriture est tout à fait valable. Si nous n'avions pas indiqué cet attribut, le son n'aurait pu être déclenché que depuis une commande JavaScript.

D'autres attributs sont disponibles pour la balise **<audio>**, tous étant booléens et pouvant donc s'utiliser sans valeur ou en répétant leur nom dans une chaîne de caractères pour être compatible avec la norme XHTML :

- **autoplay** : le fichier audio est lu dès que la page est chargée ;
- **loop** : le son sera joué en boucle. Attention : si vous n'utilisez pas de contrôles JavaScript, cet attribut doit forcément être associé à **controls** ou à **autoplay** pour avoir un impact ;
- **muted** : le volume est automatiquement passé en mode **mute**.

Je n'ai pas parlé de l'attribut **preload**, qui permet de simplement télécharger le fichier audio pour qu'il soit disponible pour une action future. Cet attribut peut prendre plusieurs valeurs :

- **auto** : pré-chargement automatique. C'est la valeur par défaut lorsque l'on ne donne pas de valeur explicite à l'attribut **preload** ;
- **metadata** : pré-chargement des métadonnées liées au fichier uniquement ;
- **none** : pas de pré-chargement... Peut-être est-il plus simple de ne pas utiliser l'attribut **preload** dans ce cas ?

Notez que si l'attribut **autoplay** est utilisé, alors l'attribut **preload** sera ignoré.

Enfin, l'attribut **src** permet de spécifier l'URL du fichier son... Mais on ne précisera ici qu'un seul format de fichier et il est donc préférable de passer par les balises **<source>**.

Pour tester ces attributs tout en nous plaçant dans un cadre de développement de jeu, nous allons créer une page comportant deux « boutons » : le premier bouton permettra de désactiver/activer une musique d'ambiance exécutée en boucle et le second permettra de lancer un son (personnage ouvrant une porte, sautant, explosion, etc.). La première partie du code sera le fichier HTML permettant de charger les fichiers audio :

```

01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test de la balise audio</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="script/audio_control.js"></script>

```

```

08: </head>
09:
10: <body>
11:   <audio id="bg_music" preload loop>
12:     <source src="night.mp3" type="audio/mp3"/>
13:     <source src="night.ogg" type="audio/ogg"/>
14:     <source src="night.aac" type="audio/aac"/>
15:     Votre navigateur ne supporte pas la balise audio!
16:   </audio>
17:
18:   <audio id="open_door" preload>
19:     <source src="door.mp3" type="audio/mp3"/>
20:     <source src="door.ogg" type="audio/ogg"/>
21:     <source src="door.aac" type="audio/aac"/>
22:     Votre navigateur ne supporte pas la balise audio!
23:   </audio>
24:
25:   <div id="toggle_bg_music">Toggle background music</div>
26:   <div id="start_open_door">Open door</div>
27:   <div id="control"></div>
28: </body>
29: </html>

```

Les fichiers de script sont chargés dans les lignes 6 et 7. Vous noterez que nous utiliserons le framework jQuery et notre fichier de contrôle des fonctions audio se nommera **audio_control.js**. Les lignes 11 à 16 définissent le chargement du premier fichier de musique dans les trois formats MP3, Ogg et AAC avec un pré-chargement automatique (**preload**) et un mode de boucle infinie activé (**loop**). Les lignes 18 à 23 définissent le chargement du second fichier son, qui lui, sera simplement pré-chargé.

Les lignes 25 et 26 nous permettent de définir deux **<div>** sur lesquels nous activerons la détection du clic de souris pour en faire des « boutons ». Le premier bouton (ligne 25) permettra de lancer ou d'arrêter la musique de fond et le second bouton (ligne 26) permettra de produire un son. Enfin, la ligne 27 définit un **<div>** vide, qui permettra d'afficher des messages de contrôle du déroulement des actions audio.

Ce fichier seul ne sert à rien, passons au code du fichier JavaScript **audio_control.js** :

```

01: $(document).ready(function()
02: {
03:   var bg_music = document.getElementById('bg_music');
04:   $('#toggle_bg_music').click(function()
05:   {
06:     if (bg_music.currentTime == 0)
07:     {
08:       $('#control').html('Starting background music');
09:       bg_music.play();
10:     }
11:     else
12:     {
13:       $('#control').html('Stopping background music');
14:       bg_music.pause();
15:       bg_music.currentTime = 0;
16:     }
17:   });
18: );
19:

```

```

20: var open_door = document.getElementById('open_door');
21: $('#start_open_door').click(function()
22: {
23:   $('#control').html('Starting open door');
24:   open_door.play();
25: })
26: );
27: }
28: );

```

Le code est ici très classique, avec une fonction principale qui est appelée au chargement complet de la page (début en ligne 1 et fin en ligne 28).

Les lignes 4 à 18 permettent de définir l'action clic sur le « bouton » **toggle_bg_music**. Si, comme le teste la ligne 6, la tête de lecture se trouve au début du morceau (temps de lecture de 0 seconde), alors on affiche un petit message (ligne 8), et on lance le fichier de musique de la balise audio identifiée par **bg_music** (ligne 9), dont on a récupéré le nœud en ligne 3. Sinon, on affiche un petit message (ligne 13), on met le morceau en pause (ligne 14), et on rembobine la tête de lecture au début du fichier (ligne 15).

La gestion de l'action clic sur le bouton **start_open_door** est définie dans les lignes 20 à 26, avec simplement l'affichage d'un message dans le **<div>** identifié par **control** (ligne 23) et le lancement du fichier son en ligne 24.

Vous aurez noté que jQuery n'est pas employé pour récupérer les nœuds des éléments audio. En effet, si vous exécutez le code suivant, vous vous apercevrez que les méthodes **\$()** et **document.getElementById()** ne renvoient pas le même objet (valeur de retour **false**) : **\$('#open_door') == document.getElementById('open_door')**; Donc, dans le cadre de l'utilisation des éléments audio, il vaut mieux se passer de jQuery sous peine d'avoir des surprises...



Fig. 1 : Barre de contrôle des fichiers sons

2. Créer des effets en JavaScript

Nous savons maintenant contrôler le démarrage et l'arrêt de fichiers sons depuis JavaScript. Mais on peut aussi définir des effets à partir d'un script. Il est ainsi possible de modifier le volume d'un son lorsqu'un personnage s'approche de la source sonore, ou encore de ne pas lancer le même son plusieurs fois de suite comme nous l'avons évoqué en introduction.

2.1 Modification du volume

La modification du volume sonore se fait à l'aide de l'attribut **volume**, dont la valeur est un flottant pouvant varier de 0 à 1, 1 représentant 100 % et donc le volume maximal.

En fonction de différents tests, le volume sonore de l'ouverture de la porte de notre exemple précédent pourra être ainsi réglé par :

```
// ...
var open_door = document.getElementById('open_door');
// ...
open_door.volume = 0.3;
// ...
```

2.2 Effets sonores différents pour une même action

Supposons que nous disposions de trois sons différents correspondant à l'ouverture d'une porte. Pour établir un roulement, il faudra stocker les nœuds audio dans un tableau, puis tirer un nombre aléatoire compris entre 1 et 3 (attention, dans un tableau le premier indice est toujours 0). Dans le code HTML, nous aurons le chargement des trois fichiers sons et le bouton de déclenchement du son :

```
// ...
<script src="script/audio_control.js"></script>
// ...
<audio id="open_door_1" preload>
  <source src="door_1.ogg" type="audio/ogg"/>
  // ...
</audio>

<audio id="open_door_2" preload>
  <source src="door_2.ogg" type="audio/ogg"/>
  // ...
</audio>

<audio id="open_door_3" preload>
  <source src="door_3.ogg" type="audio/ogg"/>
  // ...
</audio>

<div id="start_open_door">Open door</div>
<div id="control"></div>
// ...
```

Le comportement est géré dans le code JavaScript :

```
01: $(document).ready(function()
02: {
03:   var doors = new Array(
04:     document.getElementById('open_door_1'),
05:     document.getElementById('open_door_2'),
06:     document.getElementById('open_door_3')
07:   );
08:
09:   $('#start_open_door').click(function()
10:   {
11:     var num_sound = Math.floor((Math.random()*3)+1);
12:     $('#control').html('Starting open door sound n.' +
num_sound);
13:     doors[num_sound-1].play();
14:   });
15: }
16: );
17: );
```

Les lignes 3 à 7 permettent de stocker les nœuds audio dans le tableau **doors** : le son identifié par **open_door_1** est stocké à l'indice 0, et ainsi de suite.

Lors du clic sur le bouton d'ouverture de la porte, nous commençons par tirer un nombre aléatoire entre 1 et 3 (ligne 11).

Nous affichons ensuite un message indiquant quel son est joué (ligne 12), puis en ligne 13, nous déclenchons la musique grâce à la méthode **play()** appliquée au nœud tiré au sort.

3. Les sprites audio

Vous connaissez les *sprites* CSS [4] ? Il s'agit d'une image contenant une multitude de petites images, qui sont sélectionnées pour l'affichage à l'aide de leur position. L'avantage de cette technique est de ne charger qu'un seul fichier et d'obtenir ainsi un chargement plus rapide. La même technique peut être appliquée aux fichiers audio !

Nous avons utilisé précédemment l'attribut **currentTime** pour rembobiner la tête de lecture et cet attribut va nous servir ici à définir où doit commencer le son que l'on souhaite jouer. Considérons que nous possédons un fichier **open_doors.ogg** contenant trois sons d'ouverture de porte les uns à la suite des autres. Pour lire des plages particulières du fichier, nous pouvons explorer plusieurs méthodes.

L'attribut **duration** indiquant la durée totale du son est en lecture seule... Dommage, cette solution aurait été la plus simple.

L'attribut **buffered** renvoie un objet de type **TimeRange** indiquant les différents segments de lecture du fichier... Mais celui-ci est également en lecture seule.

Enfin, on peut spécifier directement dans l'URI de la source audio une plage [5] à l'aide de la notation **#t=debut,fin**, comme dans **mon_son.ogg#t=10,14**. Le problème de cette méthode est qu'il faudrait modifier l'attribut **src** et nous ne l'utilisons pas de manière à être compatible avec le plus de navigateurs possibles.

Il ne nous reste plus qu'une seule solution : à chaque événement **timeupdate** (mise à jour de **currentTime**), nous allons vérifier si nous sommes arrivés au bout de la plage à jouer ou pas. Voici le code complet permettant d'implémenter des sprites audio en commençant par le code HTML :

```
01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test des sprites audio</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="script/audio_control.js"></script>
08:   </head>
09:
10:   <body>
11:     <audio id="open_doors" preload>
12:       <source src="doors.mp3" type="audio/mp3"/>
13:       <source src="doors.ogg" type="audio/ogg"/>
14:       <source src="doors.aac" type="audio/aac"/>
15:       Votre navigateur ne supporte pas la balise audio!
16:     </audio>
17:
18:     <div id="start_open_door_1">Open door 1</div>
19:     <div id="start_open_door_2">Open door 2</div>
20:     <div id="start_open_door_3">Open door 3</div>
21:     <div id="control"></div>
22:   </body>
23: </html>
```

Et le code JavaScript :

```

01: function stop_sound(event)
02: {
03:   if (event.target.currentTime > event.target.endTime)
04:   {
05:     event.target.pause();
06:     event.target.removeEventListener('timeupdate', stop_sound, false);
07:   }
08: };
09:
10: function open_door(sound, interval, num_sound)
11: {
12:   sound.currentTime = interval[0];
13:   sound.endTime      = interval[1];
14:   $('#control1').html('Starting open door sound n.' + num_sound);
15:   sound.addEventListener('timeupdate', stop_sound, false);
16:   sound.play();
17: };
18:
19: $(document).ready(function()
20: {
21:   var doors = document.getElementById('open_doors');
22:   var doors_sprite = new Object(
23:   {
24:     'old door' : [ 0, 2.2 ],
25:     'metal door' : [ 3, 3.5 ],
26:     'wood door' : [ 4, 5.3 ]
27:   });
28: );
29:
30: $('#start_open_door_1').click(function()
31: {
32:   open_door(doors, doors_sprite['old door'], 1);
33: });
34:
35:
36: $('#start_open_door_2').click(function()
37: {
38:   open_door(doors, doors_sprite['metal door'], 2)
39: });
40:
41:
42: $('#start_open_door_3').click(function()
43: {
44:   open_door(doors, doors_sprite['wood door'], 3)
45: });
46: );
47: );
48: );

```

Ce code est un peu complexe et mérite d'être commenté en détail. Tout d'abord, au chargement de la page, les lignes 19 à 48 sont exécutées : définition du nœud audio **doors** et définition des plages de son dans les lignes 22 à 28 (table de hachage contenant en clé le nom du son et en valeur un tableau désignant le début et la fin du morceau à jouer en secondes).

Les lignes 30 à 34, 36 à 40 et 42 à 46 définissent les actions à exécuter lors du clic sur un bouton : appeler la fonction **open_door()** en lui passant en paramètre le nœud audio, la plage son à jouer et un entier qui servira d'affichage de contrôle.

La fonction **open_door()** des lignes 10 à 17 déplace la tête de lecture au début de la plage (ligne 12) et stocke la fin de la plage en ligne 13. Attention : l'attribut **endTime** n'est pas standard ! C'est un attribut ajouté pour pouvoir stocker une valeur. On aurait tout aussi bien pu l'appeler **toto**. La ligne 14 permet d'afficher notre traditionnel petit message de contrôle et la ligne 15 lance la détection de l'événement **timeupdate** avec appel de la fonction **stop_sound()**. En ligne 16, nous jouons le morceau dont la fin sera régulée par la fonction **stop_sound()** des lignes 1 à 8. Si la plage son a

été entièrement jouée (test en ligne 3), alors on arrête de jouer de la musique (ligne 5) et on supprime la détection de l'événement **timeupdate** (ligne 6).

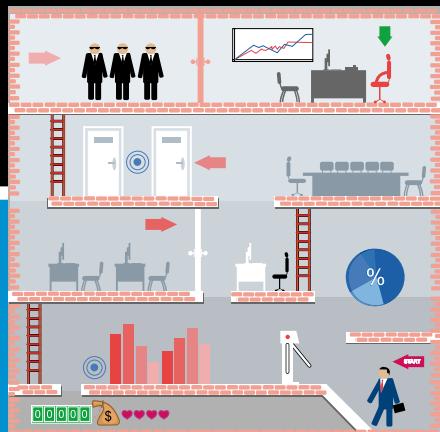
La complexité de ce code provient essentiellement de la gestion des événements avec appel à une fonction nécessitant des paramètres externes. La ruse consiste à passer ici ces paramètres directement dans l'objet audio (attribut **endTime**), qui déclenche l'événement et auquel on a donc accès par le biais de **event.target**. Pour rappel, si vous utilisez une fonction anonyme pour définir une détection d'événement avec **addEventListener()**, vous ne pourrez pas détruire la détection puisque vous ne posséderez plus de pointeur sur la fonction.

4. Conclusion

L'API Audio définie par le W3C [6] est encore à l'état de brouillon et de nombreuses fonctionnalités n'ont pas encore été implémentées par les différents navigateurs (pour l'instant disponible uniquement sur Google Chrome et Safari). Nous pourrons ainsi dans un futur proche coder simplement une atténuation du son due à la distance, un effet Doppler, un effet de tunnel, etc. Cet article nous a donc permis d'apprivoiser la gestion du son en HTML 5, mais il faudra surveiller très attentivement les améliorations futures. ■

Références

- [1] Site officiel de SoundConverter : <http://soundconverter.org/>
- [2] Site officiel d'Audacity : <http://audacity.sourceforge.net/?lang=fr>
- [3] Définition de la balise **<audio>** du W3C : <http://www.w3.org/TR/html5/the-audio-element.html#audio>
- [4] COLOMBO (Tristan), « Sprites CSS : découpe d'images en CSS », *Linux Pratique* n°67, Septembre 2011, p. 52 à 55.
- [5] Recommandation sur les fragments de média du W3C : <http://www.w3.org/TR/media-frags/>
- [6] Recommandation du W3C sur l'API Audio (brouillon) : <https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html>



ÉDITEUR DE NIVEAUX

par Sébastien Chazallet

Lorsque l'on conçoit un jeu d'arcade ou un jeu isométrique, la définition du plateau de jeu ou des décors peut rapidement devenir fastidieuse lorsqu'ils deviennent grands. Il devient alors nécessaire de faire appel à un éditeur de niveaux.



1. Définition d'un plateau de jeu

On va prendre l'exemple du plateau de jeu réalisé pour l'article « Créez un jeu complet, avec carte isométrique », que vous trouverez en page 80.

Pour réaliser ce plateau de jeu, on chargeait les tuiles et on définissait le plateau de jeu en fonction du tableau des images :

```

01 :// Définition des images composant le plateau de jeu
02 :var tuiles_filenames = Array("eau.png","desert.
png","prairie.png","montagne.png");
03 :
04 :// Chargement effectif des images
05 :var tuiles_images = new Array();
06 :for (var i=0;i<tuiles_filenames.length;i++) {
07 :    tuiles_images[i] = new Image();
08 :    tuiles_images[i].src = tuiles_filenames[i];
09 :}
10
11 :// Définition du plateau de jeu
12 :var plateau = Array(
13 :    [0,0,1,2,2,2,2,3,3,3,3],
14 :    [0,0,1,2,2,2,2,2,3,3,3],
15 :    [0,0,1,1,2,2,2,2,0,3,3,3],
16 :    [0,0,1,1,2,2,2,0,0,0,3,3],
17 :    [0,0,0,1,1,2,2,2,0,2,3,3],
18 :    [0,0,0,0,1,1,2,2,2,2,3,3],
19 :    [0,0,0,0,0,1,1,2,2,2,2,3],
20 :    [0,0,0,0,0,0,0,0,0,0,0,3],
21 :    [0,0,0,0,0,1,1,2,2,3,0,3],
22 :    [0,0,0,1,1,1,2,2,2,3,0,0],
23 :    [0,0,0,1,1,1,1,1,2,2,3,3],
24 :    [0,0,0,1,1,0,1,1,1,2,3,3]
25 :);

```

Dans notre cas, il ne s'agit que d'un plateau de jeu de 12 cases par 12 ne contenant que 4 tuiles et pourtant, il n'est pas très lisible. Si l'on souhaitait réaliser plusieurs niveaux, il faudrait créer successivement plusieurs tableaux tels que celui-ci et si les tableaux grandissaient, il deviendrait relativement fastidieux de les créer.

L'idée est donc de faire appel à une application pour créer graphiquement les cartes isométriques, comme les décors des jeux d'arcade. L'article sur les outils HTML 5 (p. 64) a présenté Tiled et la manière de l'utiliser.

Il faut noter que le site officiel présente un tutoriel extrêmement bien fait et que pour les jeux d'arcade, il est nécessaire de déclarer les tuiles dites *solides*, c'est-à-dire sur lesquelles il n'est pas possible de passer au travers.

L'article sur les outils présente la manière de charger la carte à l'aide du framework [melon.js](#). Il vous est proposé de vous y référer pour cette étape.

2. Personnages et autres éléments du game design

Lorsque l'on pense éditeur de niveaux, on pense bien entendu prioritairement à la conception de la carte isométrique ou au décor du jeu d'arcade, mais les autres éléments importants du *game design* ne sont pas en reste.

Le framework [melon.js](#) permet de prendre en compte ces différents aspects. Il est en effet possible de décrire des entités en détaillant chaque aspect :

```

01 :var PlayerEntity = me.ObjectEntity.extend( {
02 :    init: function (x, y, settings) {
03 :        this.parent(x, y , settings);
04 :        this.setVelocity(3, 15);
05 :        this.updateColRect(8,48, -1,0);
06 :        me.game.viewport.follow(this.pos, me.game.
viewport.AXIS.HORIZONTAL);
07 :    },
08 :    update : écouter
09 :});

```

Si l'on reste sur l'exemple du jeu isométrique, la méthode `écouter` serait inchangée à l'exception du fait que l'on ne se préoccupera que du calcul des positions et pas de l'appel à la méthode permettant de redessiner le plateau, car cette tâche est dévolue au framework lui-même.

De plus, pour savoir si l'on n'est pas tombé dans un piège ou dans les bras de l'ennemi, il suffit d'utiliser cette méthode :

```
01 :me.game.collide(this);
```

Encore une fois, je vous renvoie au tutoriel officiel pour aller plus loin sur ce sujet. Il est découpé en plusieurs étapes dont chacune est présente dans le dépôt Git.

Enfin, voici ce qu'il convient d'ajouter dans la méthode **Loaded** de l'objet application **jsApp** :

```
loaded: function () {
    me.state.set(me.state.PLAY, new PlayScreen());
    me.entityPool.add("mainPlayer", PlayerEntity);
    me.input.bindKey(me.input.KEY.LEFT, "left");
    me.input.bindKey(me.input.KEY.RIGHT, "right");
    me.input.bindKey(me.input.KEY.X, "jump", true);
```

Enfin, il est également possible d'ajouter du son :

```
01 :var g_ressources= [
02 :  [...]
03 :  {name: "cling", type: "audio", src: "data/audio/", channel : 1},
04 :  {name: "jump", type: "audio", src: "data/audio/", channel : 1},
05 :];
```

Le nom de la ressource est le lien qui permettra de déclencher le lancement du son.

3. Conclusion

La prise en main du framework **melon.js** nécessite un peu de temps, en particulier lorsque l'on souhaite utiliser des fonctionnalités avancées, mais la séparation entre chaque entité et la simplification d'un jeu qui en résulte, permet de gagner par la suite beaucoup de temps et d'enchaîner rapidement le développement de plusieurs niveaux de jeu. Associé au logiciel Tiled, on a là un duo gagnant parfaitement adapté à la situation. A vos claviers donc ! ■



LINUX ESSENTIEL HORS-SÉRIE N°2

INKSCAPE

LE GUIDE COMPLET POUR
MAÎTRISER LA RÉFÉRENCE DU
DESSIN VECTORIEL LIBRE !

TOUJOURS
DISPONIBLE
SUR : www.ed-diamond.com





Des outils pour ACCÉLÉRER LE DÉVELOPPEMENT

par Tristan Colombo

Jusqu'à présent nous avons décortiqué les différentes étapes de la création d'un jeu pour bien comprendre les techniques employées pour les différentes tâches comme l'animation des sprites, la création des écrans, etc. Il est maintenant temps d'accélérer nos développements...



Dans cet article, je vais vous présenter trois outils qui me paraissent intéressants : une extension pour éditeur de code permettant d'ajouter de nombreux raccourcis pour écrire votre code HTML ou CSS, une bibliothèque JavaScript dédiée à la création de jeux et un moteur physique permettant de modéliser des comportements réels.

1. Zen-Coding

Zen-Coding est un projet permettant d'écrire du code HTML grâce à de nombreux raccourcis que les adeptes de jQuery n'auront pas à apprendre puisqu'ils utilisent la même syntaxe. Cette extension est disponible pour Eclipse [1], mais également pour Vim [2] et Emacs [3]. Je ne développerai ici que son installation sous Vim, l'utilisation étant ensuite la même quel que soit l'éditeur.

Sous Vim, l'installation d'extensions est grandement facilitée par une autre extension : Pathogen. Cette extension devra être démarrée automatiquement et elle aura pour fonction de charger les extensions que nous aurons placées dans un répertoire spécial, le répertoire **bundle**.

Nous devons donc créer dans le répertoire de configuration **~/.vim** deux sous-répertoires : **autoload** pour y stocker Pathogen et **bundle** pour les autres extensions :

```
login@server:~$ mkdir ~/.vim/{autoload,bundle}
```

Pathogen doit ensuite être installé dans le répertoire **autoload** grâce à une commande **curl** (si vous ne disposez pas de **curl**, vous pouvez installer cette commande directement depuis les dépôts de votre distribution) :

```
login@server:~$ curl -so ~/.vim/autoload/pathogen.vim https://raw.github.com/tjope/vim-pathogen/HEAD/autoload/pathogen.vim
```

Pour démarrer Pathogen au lancement de Vim, nous devons ajouter les lignes suivantes au fichier **~/.vim/vimrc** :

```
01: " Chargement de Pathogen
02: call pathogen#runtime_append_all_bundles()
03: call pathogen#helptags()
```

Pathogen ira dorénavant scruter le répertoire **~/.vim/bundle** pour lancer toutes les extensions qui s'y trouvent... Nous pouvons donc y installer ZenCoding-Vim :

```
login@server:~$ git clone https://github.com/matttn/zencoding-vim.git bundle/zencoding-vim
```

Il ne reste plus qu'à utiliser l'extension dont quelques exemples d'utilisation vont suivre. Après chaque séquence de raccourcis que vous taperez en mode insertion, appuyez sur les touches **[Ctrl]+[y]+[,]** pour que votre code soit « étendu ».

Le squelette d'un fichier HTML 5 est obtenu en tapant **html:5** qui sera « traduit » en :

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
</body>
</html>
```

Pour déterminer l'identifiant d'un tag ou la valeur de son attribut **class**, il faut utiliser la notation jQuery : **#** pour indiquer l'identifiant et le point pour donner le nom de la classe. Par exemple, la chaîne **div#box.center** produit l'affichage :

```
<div id="box" class="center"></div>
```

Si l'attribut **class** possède plusieurs noms de règles CSS, il vous suffira de les enchaîner à l'aide du point, comme par exemple dans **div#box.center.red.bold**. Ceci générera le code :

```
<div id="box" class="center red bold"></div>
```

Pour définir la valeur d'autres attributs, il faudra les signaler entre crochets et donner leur valeur à l'aide du signe égal. Par exemple, **img[src="tux.png" alt="Petit pingouin"]** est « étendu » en :

```

```

Pour déterminer une hiérarchie dans des tags, il faudra utiliser les symboles **>** pour descendre d'un niveau et **<** pour remonter d'un niveau. Exemple : **ul#maListe>li#item_1>img[src="tux.png" alt="Petit pingouin"]<li#item_2**. Ceci permet de générer le code suivant :

```
<ul id="maListe">
    <li id="item_1"></li>
        <li id="item_2"></li>
</ul>
```

On peut également indiquer le nombre de tags du même type que l'on souhaite écrire (pratique pour les listes) et utiliser des compteurs grâce au caractère **\$**. Voici un exemple utilisant justement les listes : **ul>li.item_\$\$*5**. Le nombre d'occurrences du caractère **\$** indique le nombre de chiffres à utiliser pour écrire le compteur. Le code produit est :

```
<ul>
    <li class="item_01"></li>
    <li class="item_02"></li>
    <li class="item_03"></li>
    <li class="item_04"></li>
    <li class="item_05"></li>
</ul>
```

Pour commenter/dé-commenter simplement des blocs de code, placez-vous dans le tag souhaité, puis appuyez sur **[Ctrl]+[y]** puis sur **[/]**.

Ça fonctionne aussi avec les feuilles de style ! Par exemple, **bgc** devient **background-color**, **ff** devient **font-family**, **d:i** pour **display : inline;**, etc. Bref, voici un outil permettant de gagner énormément de temps en codages diverses !

2. melonJS

melonJS fait partie des nombreux frameworks de développement de jeux en JavaScript. Il est pour moi le plus intéressant et possède l'avantage d'être distribué sous licence

MIT. De plus, vous n'aurez pas à créer votre propre éditeur de niveaux, puisqu'il est capable de lire les fichiers issus de l'éditeur **Tiled Map Editor**. Voyons comment installer ces outils et les utiliser.

2.1 Installation

melonJS s'installe très simplement grâce au système de gestion de versions concurrentes Git. Si cet outil n'est pas présent sur votre système, vous pouvez l'installer depuis les dépôts de votre distribution (exemple pour des distributions basées sur Debian) :

```
login@server:~$ sudo aptitude install git
```

Placez-vous ensuite dans le répertoire d'installation et tapez :

```
login@server:~$ git clone https://github.com/obiot/melonJS.git
```

Attention, il ne s'agit pas d'un simple fichier JavaScript, d'autres manipulations sont encore nécessaires. Assurez-vous que l'utilitaire **make** et le langage Java sont bien installés sur votre système, puis dans le répertoire **melonJS/** qui est apparu, créez un nouveau répertoire **build/** puis exécutez la commande **make** :

```
login@server:~$ cd melonJS
login@server:~$ mkdir build
login@server:~$ make
```

Installez ensuite Node.js et le gestionnaire de paquets **npm** (*Node Package Manager*) :

```
login@server:~$ sudo aptitude install nodejs npm
```

Installez enfin le langage CoffeeScript [5] (petit langage générant du code JavaScript) :

```
login@server:~$ sudo npm install -g coffee-script
```

Dans le répertoire **melonJS**, lancez la commande suivante qui va lire le fichier **package.json** contenant des informations sur les dépendances et qui va résoudre ces dépendances par l'installation de nouveaux paquetages Node.js :

```
login@server:~$ npm install -d
```

Il faut lancer ensuite l'équivalent du **make** pour les fichiers CoffeeScript : la commande **cake**, qui va lire le fichier **Cakefile** :

```
login@server:~$ cake build:browser
```

Et là... patatas :

```
node.js:201
    throw e; // process.nextTick error, or 'error' event
  on first tick
  ^
SyntaxError: In Cakefile, octal literal '0755' must be
prefixed with '0o' on line 51
...
...
```

Pas de panique ! Éditez le fichier `Cakefile` et modifiez la ligne 51 en :

```
fs.mkdir builddir, 0755, ->
```

Vous pouvez relancer la commande `cake build:browser`, tout fonctionnera correctement cette fois-ci.

Pour créer les fichiers de documentation (apparaissant dans le répertoire `docs/`), tapez :

```
login@server:~$ make doc
```

L'installation de melonJS est achevée ! Vous trouverez un répertoire `examples/` contenant... des exemples et un répertoire `tutorial/` contenant les fichiers nécessaires à la réalisation du tutoriel qui est très bien fait [4]. Si vous testez les exemples, je vous recommande `dirtyRect`.

Pour tirer pleinement parti de la puissance de melonJS, nous allons installer l'éditeur de niveaux Tiled Map Editor. Pour cela, vous pouvez soit vous rendre sur le site <http://www.mapeditor.org> pour récupérer et compiler le code source, soit utiliser le dépôt Ubuntu. Pour suivre la seconde méthode, il va vous falloir ajouter le dépôt :

```
login@server:~$ sudo add-apt-repository ppa:mapeditor.org/tiled
```

Après la mise à jour des dépôts, vous pourrez installer Tiled Map Editor :

```
login@server:~$ sudo aptitude update
login@server:~$ sudo aptitude install tiled
```

2.2 Utilisation de l'éditeur de niveaux Tiled Map Editor

Pour lancer l'éditeur de niveaux, exécutez la commande `tiled` dans un terminal. Vous obtiendrez l'ouverture d'une fenêtre dans laquelle la première action à faire sera de régler les préférences. Dans le menu, sélectionnez **Édition -> Préférences**, puis indiquez que vous souhaitez enregistrer les données en tant que **Base64 (non compressé)** comme le montre la figure 1. En effet, melonJS ne sait lire que ces fichiers.



Fig. 1 : Réglage du format d'enregistrement dans Tiled Map Editor



Fig. 2 : Réglage du format d'enregistrement dans Tiled Map Editor

Pour créer une nouvelle carte, il faut cliquer sur l'icône **Nouveau** (première icône en haut à gauche dans la barre d'outils). Une fenêtre apparaîtra dans laquelle vous devrez choisir l'orientation (orthogonale ou isométrique) de votre carte, la taille des sprites (**Tile size**) et la taille de la carte exprimée en nombre de sprites ou **Map size** (voir figure 2).

Nous allons utiliser les sprites de la SpritesLib qui nous avaient déjà servi pour mettre en place l'animation des sprites. Les sprites du décor ont une taille de 32 x 32 pixels. Pour y avoir accès, il faut créer un nouvel ensemble de sprites en cliquant sur **Cartes -> Nouveau Tileset**, puis en indiquant le nom du fichier contenant les sprites (voir figure 3).

Attention, le fichier `block2.png` contient un décalage de 2 pixels en abscisse et d'un pixel en ordonnée. Pour pouvoir l'utiliser simplement, je vous recommande de supprimer cette marge à l'aide de GIMP. Vous devrez ensuite indiquer un espacement de 2 pixels entre chaque sprite. Le travail s'effectue ensuite en faisant des opérations de glisser-déposer de sprites sur votre carte. Vous pouvez ajouter des calques de manière à obtenir plusieurs plans, comme le montre la figure 4. Pour finir, sauvegardez votre œuvre au format `.tmx`.

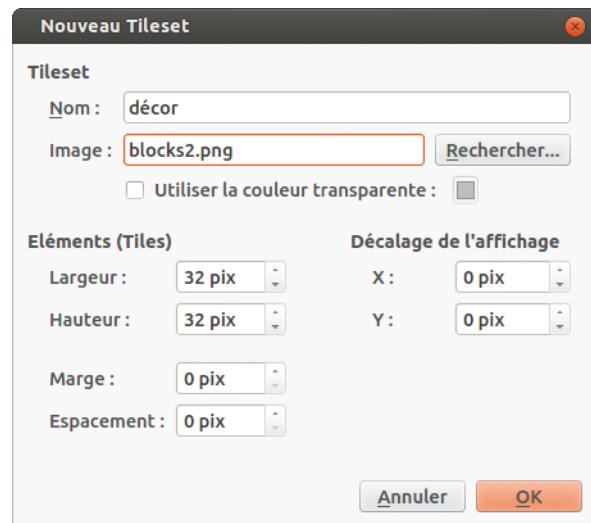


Fig. 3 : Définition d'un tileset dans Tiled Map Editor

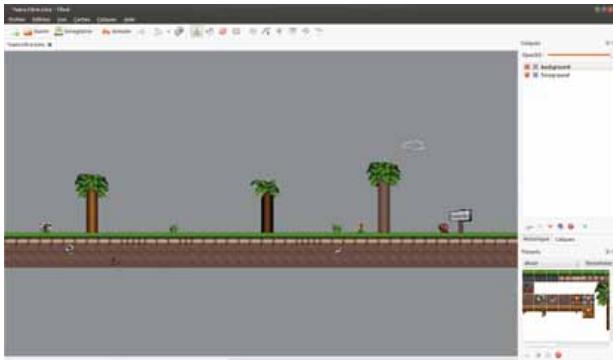


Fig. 4 : Création d'une carte sur plusieurs plans dans Tiled Map Editor

Essayons maintenant d'afficher notre niveau. La structure d'un jeu melonJS est la suivante :

```

01: <!DOCTYPE html>
02: <html>
03:   <head>
04:     <title>Affichage de niveaux</title>
05:     <meta charset="utf-8" />
06:     <script src="lib/melonJS-0.9.3-min.js"></script>
07:     <script src="main.js"></script>
08:   </head>
09:
10:   <body>
11:     <div id="jsapp">
12:     </div>
13:   </body>
14: </html>
```

Il s'agit d'un fichier HTML classique dans lequel les fichiers JavaScript de melonJS (placés dans le répertoire **lib/**) et de contrôle du jeu sont chargés dans les lignes 6 et 7. Dans le corps du document, il n'y a qu'une seule balise **<div>** identifiée par **jsapp** qui permettra d'insérer les éléments du jeu (essentiellement des **canvas**).

Le fichier **main.js** va permettre de charger les décors et de lancer la boucle du jeu. Ici, la boucle sera réduite à sa plus simple expression puisque nous ne voulons qu'afficher un écran du jeu sans aucune interaction. Les données du jeu (fichiers **tmx** et **png**) sont placées dans le répertoire **data/** :

```

01: var g_ressources = [
02:   {
03:     name: "blocks2",
04:     type: "image",
05:     src: "data/blocks2.png"
06:   },
07:   {
08:     name: "ecrans",
09:     type: "tmx",
10:     src: "data/ecrans.tmx"
11:   }
12: ];
13:
14:
15: var jsApp =
16: {
17:   onload: function()
18: }
```

```

19:           if (!me.video.init('jsapp', 640, 480, false, 1.0))
20:             {
21:               alert("Votre navigateur ne supporte pas la
balise canvas ! " +
22:                     "Essayez-en un autre !");
23:               return;
24:             }
25:
26:             me.audio.init("mp3,ogg");
27:             me.loader.onload = this.loaded.bind(this);
28:             me.loader.preload(g_ressources);
29:             me.state.change(me.state.LOADING);
30:           },
31:           loaded: function ()
32:             {
33:               me.state.set(me.state.PLAY, new PlayScreen());
34:               me.state.change(me.state.PLAY);
35:             }
36:         };
37:
38: var PlayScreen = me.ScreenObject.extend(
39:   {
40:     onResetEvent: function()
41:       {
42:         me.levelDirector.loadLevel("ecrans");
43:       },
44:     onDestroyEvent: function()
45:       {
46:       }
47:   });
48:
49:
50: window.onReady(function()
51:   {
52:     jsApp.onload();
53:   }
54: );
```

Les lignes 1 à 12 définissent les fichiers d'image et de carte. Attention : le champ **name** doit obligatoirement correspondre au nom du fichier ! Par exemple, en ligne 5 le fichier s'appelle **blocks2.png**, donc le champ **name** de la ligne 3 a pour valeur **blocks2**.

Les lignes 15 à 45 définissent un objet **jsApp** qui est l'élément principal du code. Cet objet possédera une méthode **onload()** qui sera appelée quand tous les éléments de la page HTML seront chargés (lignes 50 à 53). Revenons à l'objet **jsApp** : la méthode **onload()** est définie dans les lignes 17 à 30. On commence par initialiser les **canvas** où le jeu sera affiché (ligne 19). On indique le nom du **div** qui sera utilisé pour insérer les **canvas** et sa taille (ici 640 x 480 px). L'objet **me** qui est utilisé ici et dans la suite du code est simplement l'espace de nom pour melon engine : tous les objets et fonctions de la librairie appartiennent à cet espace de nom et seront donc préfixés par **me**.

Nous initialisons ensuite la prise en compte des fichiers audio pour la suite (ligne 26). Nous indiquons les ressources à charger (lignes 27 et 28) avant de les charger effectivement en affichant un écran possédant une barre de chargement (ligne 29). Par défaut, cet écran comporte le nom melonJS et une barre de progression verte. La méthode **loaded()** des lignes 31 à 35 est une méthode appelée lorsque tous les éléments sont chargés et qui lance la boucle de jeu. L'objet **PlayScreen** des lignes 38 à 47 définit le

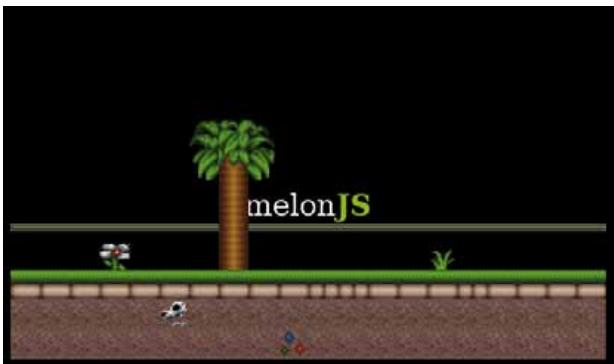


Fig. 5 : Affichage d'une partie d'un niveau avec melonJS

comportement des objets en fonction des changements d'état. Dans notre exemple, il ne contient que très peu de code puisque sa seule action est d'afficher une image fixe.

melonJS permet également d'insérer des sprites animés pour que le joueur puisse se déplacer ; vous pouvez définir un calque de collisions indiquant les parties « solides » du décor, vous pouvez affiner la boîte de collisions de votre personnage joueur, ajouter un scrolling avec un effet de parallaxe (décor arrière défilant moins vite que le décor avant), insérer de la musique, bref... vous pouvez tout faire ! Le sujet est très vaste et il faudrait y consacrer un hors-série entier ! En attendant cet hypothétique magazine, je vous recommande le tutoriel qui, même s'il laisse planer de grosses zones d'ombre, est très bien fait.

3. box2d, un moteur physique en 2D

Vous avez déjà joué à Angry Birds ? Vous vous demandez comment les éléments s'effondrant les uns sur les autres font pour avoir un comportement réaliste ? Ne cherchez plus : c'est box2d. À l'origine, cette librairie a été développée en C++, puis adaptée pour un usage Web sous le nom de box2dweb (il existe une alternative nommée box2d-js, mais elle n'est plus maintenue et contient de nombreux fichiers, ce qui la rend plus difficilement manipulable). Grâce à elle, vous pourrez modéliser la gravité, l'élasticité, des collisions, etc.

Pour pouvoir tester ce moteur, nous allons utiliser une API qui simplifie les commandes : boxbox. En effet, le portage du C++ vers le JavaScript a produit une librairie peu lisible. Pour installer boxbox (et par là même box2dweb), il suffit de taper :

```
login@server:~$ git clone https://github.com/incompli/boxbox.git
```

Au niveau de la philosophie générale de ce moteur, vous devez commencer par créer un « monde » qui contiendra les paramètres de la simulation (intensité de la gravité, fréquence d'affichage, etc.), un « corps » spécifiant les éléments statiques et dynamiques et surtout, un « sol » et des « fixtures » ou effets divers à appliquer.

Tous les éléments seront affichés à l'intérieur d'un **canvas** qu'il faudra créer dans le code HTML :

```
01: <!DOCTYPE html>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Test de boxbox</title>
06:     <script src="script/Box2dWeb-2.1.a.3.min.js"></script>
07:     <script src="script/boxbox.js"></script>
08:     <script src="script/demo.js"></script>
09:   </head>
10:
11:   <body>
12:     <canvas id="boxbox_test" width="600" height="400">
13:       Votre navigateur ne supporte pas la balise canvas
14:     </canvas>
15:   </body>
16: </html>
```

Les fichiers des librairies sont chargés dans les lignes 6 et 7, le fichier de contrôle est lui chargé en ligne 8 et le **canvas** est défini dans les lignes 12 à 14.

Encore une fois, c'est le fichier JavaScript qui va positionner et animer les éléments :

```
01: window.onload = function ()
02: {
03:   var canvas = document.getElementById('boxbox_test');
04:
05:   var world = boxbox.createWorld(canvas, {
06:     debugDraw:false });
07:
08:   var groundTemplate =
09:   {
10:     name: 'ground',
11:     type: 'static',
12:     height: .8,
13:     color: 'blue',
14:     borderColor: 'rgba(0, 0, 100, .5)',
15:     borderWidth: 3
16:   };
17:
18:   world.createEntity(groundTemplate, { width: 20, x: 10,
19: y: 13.22 });
20:
21:   world.createEntity(groundTemplate, { width: 6, x: 3,
22: y: 5 });
23:
24:   world.createEntity(
25:   {
26:     name: 'square',
27:     x: 13,
28:     y: 8,
29:     height: 1.6,
30:     width: .4,
31:     imageOffsetY: -.2
32:   });
33:
34:   world.createEntity(
35:   {
36:     name: 'tux',
37:     shape: 'square',
38:     x: 14,
39:     y: 3,
```

```

38:     height: 3,
39:     width: 3,
40:     density: .5,
41:     image: 'tux.png',
42:     imageStretchToFit: true
43:   }
44: );
45:
46: world.createEntity(
47: {
48:   name: 'poly',
49:   shape: 'polygon',
50:   x: 5,
51:   y: 8
52: }
53: );
54:
55: var platform = world.createEntity(
56: {
57:   name: 'platform',
58:   fixedRotation: true,
59:   height: .2,
60:   width: 2
61: }
62: );
63:
64: var platformMovingUp = true;
65:
66: window.setInterval(function() {
67:   platformMovingUp = !platformMovingUp;
68:   if (platformMovingUp)
69:   {
70:     platform.setVelocity('moving platform', 5, 0);
71:   }
72:   else
73:   {
74:     platform.setVelocity('moving platform', 5, 180);
75:   }
76: }, 1500);
77: };

```

Le nœud du `canvas` est récupéré en ligne 3 pour être utilisé en ligne 5 pour créer le « monde ». Nous définissons ensuite le « sol » dans les lignes 7 à 15. Il ne s'agit que d'une définition de la représentation du « sol » (couleur, épaisseur, etc.), pas de son positionnement, qui est lui effectué dans les lignes 17 et 18 (en deux parties, comme le montre la figure 6).

Les différents éléments de la scène sont ensuite créés à l'aide de la méthode `createEntity()` de l'objet `world`. Les attributs `x` et `y` donnent le positionnement de l'objet, `shape` sa forme, `density` sa densité (les objets peuvent être plus ou moins denses : en cas de chute, ils tomberont plus vite par exemple, etc.), `imageOffsetY` permet de régler le décalage sur `y`, etc.

Bien sûr, nous ne sommes pas obligés de ne dessiner que des formes géométriques, nous pouvons ajouter à chaque forme une image. C'est le cas de l'élément carré défini dans les lignes 32 à 44 et sur lequel nous avons « collé » l'image d'un pingouin (ligne 41).

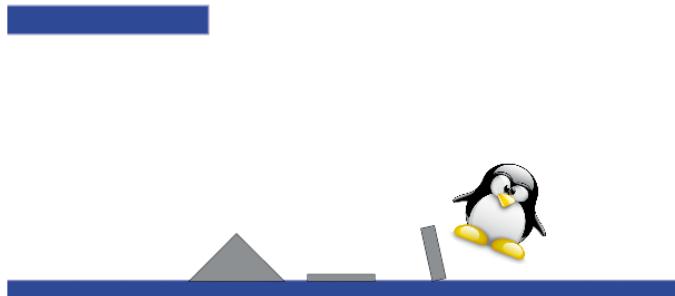


Fig. 6 : Utilisation du moteur physique box2d au travers de l'API boxbox

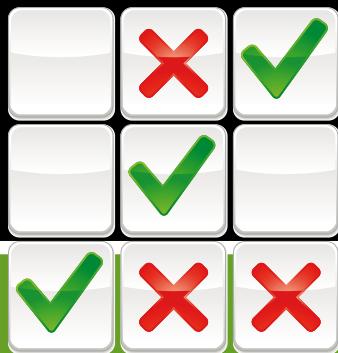
Il y a enfin un rectangle particulier car mouvant. Pour animer cette forme, il faut garder un pointeur sur cette dernière : c'est la variable `platform` de la ligne 55. L'animation se fait ensuite à l'aide de la méthode `setInterval()` et de `setVelocity()` qui prend en paramètres un identifiant, la force de la poussée (ce qui donnera la distance), et la direction en degrés (0 vers le bas et 180 vers le haut).

4. Conclusion

Les outils présentés dans cet article, en dehors de Zen-Coding, ont été conçus spécialement pour le développement de jeux. Ils sont donc optimisés pour les tâches les plus courantes, inutile de réinventer la roue (carrée ou pas, suivant les développeurs...). Il est toutefois important de bien comprendre les différents mécanismes vus au cours des articles précédents de manière à ne pas rester prisonnier de telle ou telle librairie. Si des fonctionnalités ne sont pas présentes, il faudra les ajouter et si trop de modifications doivent intervenir c'est que la librairie ne convient pas à votre projet. Cherchez-en une autre ou développez la vôtre ! ■

Références

- [1] Site officiel du projet Zen-Coding :
<http://code.google.com/p/zen-coding/>
- [2] Page Github du projet ZenCoding-Vim :
<https://github.com/mattm/zencoding-vim>
- [3] Page Github du projet Zen-Coding pour Emacs :
<https://github.com/rooney/zencoding>
- [4] Tutoriel melonJS : <http://www.melonjs.org/tutorial/>
- [5] Site officiel du langage CoffeeScript :
<http://coffeescript.org/>
- [6] Site officiel de boxbox :
<http://incompl.github.com/boxbox/>



UN JEU DE MORPION en HTML 5

par Tristan Colombo

Le jeu du morpion ou Tic-tac-toe est un jeu fort simple comportant un damier de trois fois trois cases et cinq pions par joueur. Le but est d'être le premier à parvenir à aligner trois pions en ligne, colonne ou diagonale. Ne présentant aucune difficulté particulière, ce jeu est particulièrement adapté à la mise en place des différentes techniques que nous avons pu voir tout au long de ce hors-série.



Pour coder ce jeu de Tic-tac-toe, même si les règles sont très simples, il faut d'abord avoir une idée globale de ce à quoi nous voulons arriver. Un schéma réalisé rapidement à l'aide de Pencil [1] nous permet d'avoir une vision des différents objets qui seront présents sur notre page (voir figure 1).

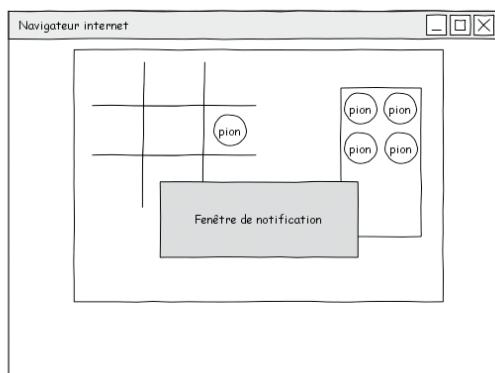


Fig. 1 : Schéma représentant l'affichage souhaité pour notre jeu

Effectuons une liste des principales parties que l'on peut dégager pour réaliser notre jeu et les technologies que nous emploierons :

- Affichage de messages en utilisant une police externe : `@font-face` ;
- Affichage des pions : sprites CSS ;
- Déplacement des pions : *drag and drop* ;
- Émission d'un son lors de la prise et du positionnement d'un pion, ainsi qu'en cas de victoire ou de défaite : balise `<audio>` et sprites audio ;
- Sauvegarde du meilleur score : `localStorage` ;
- Jeu contre l'ordinateur : websockets et socket.IO.

Bien sûr, le fait de coder le jeu de la machine sur un serveur distant n'est pas très indiqué pour ce jeu, mais il permet de tester les websockets en grande taille et l'adaptation pour un jeu à deux joueurs distants sera très simple.

Enfin, un dernier aspect technique : j'ai décidé d'employer LESS dans ce projet pour générer la feuille de styles CSS. Ceci me permettra d'introduire ce mini-langage pour ceux d'entre vous qui ne le connaissent pas.

Comme vous devez vous en douter, le code que nous étudierons ici est nettement plus important que dans les autres articles où nous pouvions vraiment isoler des cas concrets permettant d'appliquer une technique bien particulière. Je ne vous présenterai donc que des parties de code, en conservant leur numérotation originale. Pour obtenir le code complet, vous pourrez vous rendre sur <http://github.com/tcolombo/tictactoe-html5>.

1. LESS pour écrire moins de CSS

LESS [2] est un langage permettant de générer des feuilles de styles CSS à partir d'une syntaxe s'approchant fortement des CSS, mais permettant d'utiliser des variables, des fonctions, etc. On peut l'employer de deux manières possibles :

- Charger le fichier LESS et un fichier JavaScript permettant de « décoder » les instructions en CSS :

```
<link rel="stylesheet/less" type="text/css" href="style/style.less">
<script src="script/less.js"></script>
```

- Générer le fichier CSS correspondant au fichier LESS et utiliser ainsi un fichier de styles classique. Dans ce cas, il faudra installer le compilateur basé sur Node.js. L'installation se fait simplement par la commande `npm` :

```
sudo npm install -g less
```

Pour générer votre fichier de styles, vous n'aurez ensuite plus qu'à taper :

```
less style.less > style.css
```

Vous savez utiliser un fichier LESS... Mais peut-être souhaiteriez-vous maintenant avoir un aperçu de la syntaxe et de ce que l'on peut réaliser avec ? Je vous avais parlé des variables. Voici la manière de les écrire en LESS :

```
@color_red: #540000;
```

Nous avons défini ici une variable `@color_red` ayant pour valeur `#540000`. Nous aurions tout aussi bien pu définir une taille ou n'importe quelle autre valeur :

```
@size: 128px;
```

Pour utiliser ces variables, il suffit ensuite d'y faire référence dans des règles CSS :

```
.maClasse
{
    width: @size;
    color: @color_red;
}
```

Une fois compilé, l'ensemble du code précédent générera le code CSS suivant :

```
.maClasse
{
    width: 128px;
    color: #540000;
}
```

Ces variables sont déjà bien pratiques... Mais vous n'avez pas encore vu les fonctions ! Vous connaissez ces règles CSS qui ne sont pas encore stables et qui possèdent un préfixe en fonction du navigateur ? Il faut ajouter `-moz-` pour Firefox, `-webkit-` pour Google Chrome, `-o-` pour opéra, etc. Donc, au lieu d'écrire une seule règle, il faudra à chaque fois en écrire au moins quatre. Les fonctions permettent de gérer ce problème :

```
01: .rounded-corners (@radius: 5px)
02: {
03:     border-radius: @radius;
04:     -moz-border-radius: @radius;
05:     -webkit-border-radius: @radius;
06:     -o-border-radius: @radius;
07: }
08:
09: #header
10: {
11:     .rounded-corners;
12: }
13:
14: #footer
15: {
16:     .rounded-corners(10px);
17: }
```

Dans les lignes 1 à 7 nous avons défini une fonction `.rounded-corners` prenant un paramètre ayant pour valeur par défaut `5px`. Cette fonction contient toutes les écritures possibles pour la règle `border-radius`. La ligne 11 permet d'appeler cette fonction en utilisant la valeur par défaut, alors que la ligne 16 utilise une valeur de `10px`. Ce code produit le code CSS suivant :

```
#header
{
    border-radius: 5px;
    -moz-border-radius: 5px;
    -webkit-border-radius: 5px;
    -o-border-radius: 5px;
}

#footer
{
    border-radius: 10px;
    -moz-border-radius: 5px;
    -webkit-border-radius: 10px;
    -o-border-radius: 10px;
}
```

Ce sont là les principales fonctionnalités de LESS. Il en existe bien d'autres, d'un usage moins courant, que je vous invite à découvrir sur <http://lesscss.org/#docs>. Si vous utilisez Vim pour éditer votre code, vous vous rendrez compte que la coloration syntaxique ne fonctionne pas avec les fichiers LESS. L'installation de l'extension `vim-less` [3] règle le problème (faites attention de bien ajouter la commande `call pathogen#infect()` dans votre fichier `~/.vimrc` si vous utilisez l'extension `pathogen`).

2. Organisation du code

Revenons à notre projet. Nous avons une vision globale du développement et nous avons installé (presque) tous les outils dont nous aurons besoin. Il est temps de réfléchir à une architecture pour organiser nos différents fichiers. Il n'y a pas un type d'organisation obligatoire, mais nous savons que nous aurons le code du serveur, le code du client en JavaScript, des fichiers de styles, un fichier audio, un fichier image, des polices de caractères, etc. Je vous propose l'organisation suivante :

```
tictactoe-html5
├── index.html
└── media
    ├── audio
    ├── fonts
    ├── images
    └── style
├── script
└── server
    └── node-modules
```

Nous séparerons ainsi clairement la partie média (tout en cloisonnant les différents médias dans des sous-répertoires distincts), la partie serveur et la partie cliente avec le répertoire `script`.

3. La base : le fichier HTML

Le fichier HTML sera le point d'entrée dans notre jeu. C'est grâce à lui que nous allons charger les différents fichiers JavaScript et poser les balises de base définissant la fenêtre de jeu, le damier, les pions, les sons et la fenêtre de notification :

```

01: <!DOCTYPE HTML>
02: <html lang="fr">
03:   <head>
04:     <meta charset="utf-8">
05:     <title>Tic Tac Toe en html5</title>
06:     <script src="script/jquery-1.7.2.min.js"></script>
07:     <script src="http://localhost:3000/socket.io/socket.
io.js"></script>
08:     <script src="script/control.js"></script>
09:     <link rel="stylesheet" href="media/style/tictactoe.css" />
10:   </head>
11:
12:   <body>
13:     <audio id="game_sounds" preload>
14:       <source src="media/audio/sounds.mp3" type="audio/mp3"
/>
15:       <source src="media/audio/sounds.ogg" type="audio/ogg" />
16:     </audio>
17:
18:     <div id="gameZone">
19:       <div id="array">
20:         <div id="line_1">
21:           <div id="c11" class="case"></div>
22:           <div id="c12" class="case"></div>
23:           <div id="c13" class="case"></div>
24:         </div>
25:         <div id="line_2">
26:           <div id="c21" class="case"></div>
27:           <div id="c22" class="case"></div>
28:           <div id="c23" class="case"></div>
29:         </div>
30:         <div id="line_3">
31:           <div id="c31" class="case"></div>
32:           <div id="c32" class="case"></div>
33:           <div id="c33" class="case"></div>
34:         </div>
35:       </div>
36:       <div id="tokens">
37:         <div id="t_1" class="token"></div>
38:         <div id="t_2" class="token"></div>
39:         <div id="t_3" class="token"></div>
40:         <div id="t_4" class="token"></div>
41:         <div id="t_5" class="token"></div>
42:       </div>
43:       <div id="notification"></div>
44:     </div>
45:   </body>
46: </html>
```

Nous utiliserons l'incontournable jQuery (ligne 6) et vous reconnaîtrez en ligne 7 le chargement du fichier **socket.io.js**. Le fichier de gestion des événements côté client se nomme **control.js** et son chargement est effectué en ligne 8. La ligne 9 permet de charger la feuille de styles générée depuis le fichier LESS.

Les lignes 13 à 16 permettent de charger le fichier de son et de lui affecter l'identifiant **game_sounds**. Puisque nous utilisons des sprites audio, il a fallu créer ce fichier. Pour cela, je me suis servi de divers sons basiques du système (dans **/usr/share/sounds**) et du logiciel Audacity pour assembler tous les sons dans un seul fichier (en essayant de caler le début de chaque morceau sur un indice de temps à valeur entière... C'est plus simple pour la suite !).

Nous définissons ensuite la zone de jeu (identifiée par **gameZone**) dans les lignes 18 à 44. Cette zone contient le damier (lignes 19 à 35), la boîte des jetons (lignes 36 à 37) et la fenêtre de notification (ligne 43). Vous remarquerez que le damier n'est pas créé à l'aide d'une balise **<table>** : nous ne stockons pas des données dans un tableau, donc nous n'avons pas à utiliser un tableau ! La structure est un empilement de balises **<div>** qui seront mises en page par la feuille de styles. La balise identifiée par **array** englobe tout le tableau qui est composé de lignes (**line_1**, **line_2** et **line_3**), qui sont elles-mêmes composées de cellules (**c11**, **c12**, ..., **c33**). Chaque case du damier est ainsi identifiée par un nom comportant sa position en ligne et en colonne. Par exemple, pour **c12**, la case se situe en première ligne, deuxième colonne. Les jetons des lignes 37 à 41 possèdent eux aussi un système d'identification par leur nom : **t_1** à **t_5**.

Mais sans mise en forme, nous n'obtenons pas un résultat très intéressant...

4. Présentation : la feuille de styles

La feuille de styles est écrite en LESS avant d'être convertie en CSS. La partie intéressante du code est donc en LESS. Tout d'abord, la création des « boîtes » ou cadres permettant de délimiter la zone de jeu, les pions et la fenêtre de notification. Nous utiliserons des coins arrondis et nous ajouterons une ombre, deux options rendues possibles par CSS 3 :

```

01: @screen_width: 800px;
02: @screen_height: 600px;
...
24: .rounded-corners (@radius: 5px)
25: {
26:   border-radius: @radius;
27:   -webkit-border-radius: @radius;
28:   -moz-border-radius: @radius;
29: }
30:
31: .shadow (@color: @color_shadow)
32: {
33:   box-shadow: 8px 8px 12px @color;
34:   -webkit-box-shadow: 8px 8px 12px @color;
35:   -moz-box-shadow: 8px 8px 12px @color;
36: }
37:
38: .border-top (@color: #000)
39: {
40:   border-top: 2px solid @color;
41: }
...
```

```

58: .box (@color: #000, @bgcolor: #fff)
59: {
60:   .border-top(@color);
61:   .border-bottom(@color);
62:   .border-left(@color);
63:   .border-right(@color);
64:
65:   background-color: @bgcolor;
66:
67:   .shadow
68: }
69:
70: .center
71: {
72:   margin: auto;
73: }
...
97: #gameZone
98: {
99:   width: @screen_width;
100:  height: @screen_height;
101:
102:  .center;
103:  .rounded-corners;
104:  .box;
105: }

```

Nous commençons par définir des fonctions de manière à pouvoir réutiliser le même code pour nos trois cadres :

- **.rounded-corners** (lignes 24 à 29) pour les coins arrondis ;
- **.shadow** (lignes 31 à 36) pour l'ombre ;
- **.border-top** (lignes 38 à 41), **.border-bottom**, etc., pour définir l'encadrement. Ceci permettra de les réutiliser dans la définition du damier ;
- **.box** (lignes 58 à 68) qui définit une « boîte » avec ombrage, encadrée de noir et de fond blanc ;
- **.center** (lignes 70 à 73) pour aligner horizontalement les objets.

La définition de la zone de jeu des lignes 97 à 105 n'a plus qu'à préciser une taille et utiliser les différentes fonctions. Le code est plus lisible qu'avec une CSS classique et la zone de placement des pions et la fenêtre de notification seront définies à l'aide de ce même code.

Pour la gestion des sprites CSS des pions, nous utiliserons une image créée sous GIMP à partir de deux icônes issues du site <http://www.iconfinder.com> :

```

14: .tokenHuman ()
15: {
16:   background: url("../images/tokens.png") 0 0 no-repeat;
17: }
18:
19: .tokenServer ()
20: {
21:   background: url("../images/tokens.png") -@size 0 no-repeat;
22: }
...
173: .generic_token ()
174: {
175:   float: left;
176:
177:   width: @size;

```

```

178:   height: @size;
179: }
180:
181: .token
182: {
183:   .generic_token;
184:   .tokenHuman;
185: }
186:
187: .token_auto
188: {
189:   .generic_token;
190:   .tokenServer;
191: }

```

Le positionnement des sprites est déterminé par les lignes 16 et 21. Les règles s'appliquant à la définition d'un pion sont définies dans une fonction (lignes 173 à 179), puis utilisées pour définir la classe des pions du joueur humain (lignes 181 à 185) et du serveur (lignes 187 à 191).

Pour le damier, nous allons utiliser des modes d'affichage particuliers (**table**, **table_row** et **table_cell**) et afficher ainsi un « tableau » :

```

08: @size: 128px;
09: @bg_case: #c7b299;
...
12: @bg_overCase: #72f3f1;
...
110: #array
111: {
112:   display: table;
113:
114:   margin: 10px;
115: }
116:
117: #line_1, #line_2, #line_3
118: {
119:   display: table_row;
120: }
121:
122: .case
123: {
124:   display: table-cell;
125:   width: @size;
126:   height: @size;
127:
128:   margin: 0;
129:   padding: 0;
130:
131:   background-color: @bg_case;
132: }
133:
134: #c11, #c12, #c13, #c21, #c22, #c23
135: {
136:   .border-bottom;
137: }
...
154: .overCase
155: {
156:   background-color: @bg_overCase;
157: }

```

Chaque case a une taille fixée par la variable `@size` (lignes 125 et 126) qui correspond à la taille des pions que l'on va y déposer. Pour créer les lignes du damier, nous définissons des bordures sur chacune des cases, comme par exemple dans les lignes 134 à 137 pour la bordure inférieure. Lors du survol d'une case, avant d'y déposer un pion, nous changerons la couleur de fond de la case si cette dernière n'est pas occupée. Ceci sera réalisé par la classe `overCase` des lignes 154 à 157.

Pour incorporer une police externe, j'ai sélectionné un `@font-kit` sur le site fontsquirrel.com et placé les fichiers dans `media/fonts`. Il suffit ensuite de déclarer la règle `@font-face` et d'utiliser la police dans le document HTML :

```

77: @font-face
78: {
79:   font-family: 'QikkiRegRegular';
80:   src: url('../fonts/Qarmic_sans_Abridged-webfont.eot');
81:   src: url('../fonts/Qarmic_sans_Abridged-webfont.
eot?#iefix') format('embedded-opentype'),
82:     url('../fonts/Qarmic_sans_Abridged-webfont.woff')
format('woff'),
83:     url('../fonts/Qarmic_sans_Abridged-webfont.ttf')
format('truetype'),
84:     url('../fonts/Qarmic_sans_Abridged-webfont.
svg#QikkiRegRegular') format('svg');
85:   font-weight: normal;
86:   font-style: normal;
87: }
88:
89: body
90: {
91:   font-family: 'QikkiRegRegular', Arial, serif;
92:   font-size: 14pt;
93: }
```

```

120:           ],
121:           cpt: 0
122:         };
123:
124:         socket.on('disconnect', function ()
125:         {
126:           console.log('">>> Perte de la connexion avec le
joueur ' + socket.id);
127:         }
128:       );
129:
130:         socket.on('played', function (data)
131:         {
132:           console.log('">>> Jeu en (' + data.row + ', ' +
data.col + ') pour ' +
133:             'le joueur ' + socket.id);
134:           if (game.array[data.row][data.col] != 0)
135:           {
136:             socket.emit('cheater');
137:           }
138:           else
139:           {
140:             playHuman(game, data.row, data.col, socket);
141:             displayGame(game);
142:             playServer(game, socket);
143:           }
144:         }
145:       );
146:     }
147:   );
```

C'est le serveur qui gardera en mémoire la disposition des pions (pour limiter la triche). Les données seront stockées dans un tableau à deux dimensions qui reproduira le damier (lignes 115 à 122). Ce tableau se trouve dans un objet contenant également le nombre de coups joués. C'est à partir de ce nombre que nous pourrons sauvegarder les meilleurs scores : plus le nombre de coups est faible, meilleur est le score.

C'est toujours le joueur humain qui commence la partie. Il enverra un message étiqueté `played` contenant en données la position de son coup (`data.row` et `data.col`). Soit le coup est non valide, il y a eu triche, et on renvoie au client le message `cheater` (lignes 134 à 137), soit le coup est valide et on le répercute sur le damier du serveur (ligne 140) avant d'afficher ce damier (ligne 141) et de faire jouer le serveur (ligne 142). Comme les données proviennent directement du client, il est très simple de modifier un coup et c'est pour cela que ces données sont conservées sur le serveur.

Dans le tableau, la présence d'un `0` indique une case libre, un `1` marque la présence d'un pion du joueur humain et un `-1` indique un pion du serveur. Ce système permet de détecter simplement si un joueur a pu aligner ses pions en ligne, colonne ou diagonale en effectuant la somme de pions sur chacun des alignements possibles. Si la somme est 3, alors le joueur humain a gagné, et si la somme est -3, alors c'est le serveur qui a gagné :

```

21: function testWinner(game, player, socket)
22: {
23:   var winValue = 3 * player;
24:   var sum;
25:
26:   if (game.cpt < 5)
27:   {
28:     return false;
29:   }
30:
31:   // Test des lignes
32:   for (var row=0; row<3; row++)
33:   {
34:     sum = 0;
35:     for (var col=0; col<3; col++)
36:     {
37:       sum += game.array[row][col];
38:     }
39:     if (sum == winValue)
40:     {
41:       socket.emit('endgame', { 'status': player, 'score' : game.cpt });
42:       return true;
43:     }
44:   }
...
61:   // Test des diagonales
62:   sum = game.array[0][0] + game.array[1][1] + game.array[2][2];
63:   if (sum == winValue)
64:   {
65:     socket.emit('endgame', { 'status': player, 'score' : game.cpt });
66:     return true;
67:   }
68:   sum = game.array[2][0] + game.array[1][1] + game.array[0][2];
69:   if (sum == winValue)
70:   {
71:     socket.emit('endgame', { 'status': player, 'score' : game.cpt });
72:     return true;
73:   }
74:
75:   // Match nul
76:   if (game.cpt == 9)
77:   {
78:     socket.emit('endgame', { 'status': 0 });
79:     return true;
80:   }
81:
82:   return false;
83: }
```

En cas de victoire, de défaite ou d'égalité, nous renvoyons un message **endgame** au client contenant l'identifiant du joueur ayant remporté la partie, ainsi que le nombre de coups joués (lignes 41, 65, 71 et 78).

Au niveau du jeu du serveur, le but de cet article n'étant pas de développer la mise en pratique d'algorithme de jeu,

je n'ai pas utilisé d'algorithme particulier (type min-max), mais un simple tirage aléatoire. Certes, l'ordinateur joue de manière très stupide, mais il est du coup beaucoup plus simple de tester tous les cas...

```

03: function randint(min, max)
04: {
05:   return Math.round(min + Math.random() * (max - min))
06: };
...
92: function playServer(game, socket)
93: {
94:   var col;
95:   var row;
96:
97:   do
98:   {
99:     col = randint(0, 2);
100:    row = randint(0, 2);
101:  } while (game.array[row][col] != 0);
102:
103: game.array[row][col] = -1;
104: game.cpt++;
105: socket.emit('server', { row: row + 1, col: col + 1 });
106: return testWinner(game, -1, socket);
107: };
```

Tant que l'on ne trouve pas de case vide, on tire au sort un numéro de colonne et un numéro de ligne (lignes 97 à 101). Nous sommes assurés de sortir de la boucle, car il existe au moins une case libre, le test ayant été fait après que le joueur humain ait joué. La fonction de tirage aléatoire des lignes 3 à 6 est intéressante, car il s'agit d'une généralisation permettant de tirer un entier entre deux valeurs données. **Math.random()** renvoie un réel compris entre **0** et **1**. Donc, en le multipliant par l'écart entre la valeur maximale et la valeur minimale, on obtient un réel compris entre **0** et **(max - min)**. En y ajoutant la valeur **min**, notre réel se trouve compris entre **min** et **max**. Il ne reste plus qu'à en prendre la partie entière avec **Math.round()**.

6. Le client

Le fichier **control.js** est le plus long, car c'est lui qui va devoir gérer le plus de choses. Il doit tout d'abord se connecter au serveur. Mais que se passe-t-il si le serveur n'est pas actif ? Allez-vous laisser le joueur partir vers un jeu bien moins abouti et ludique que notre morpion pour un simple problème de connexion ? Non, mais pour cela, il va falloir gérer la reconnexion :

```

19: function reconnect_server()
20: {
21:   $.getScript('http://localhost:3000/socket.io/socket.io.js');
22:   connect_server();
23: };
24:
25: function connect_server()
```

```

26: {
27:   var socket = null;
28:
29:   try
30:   {
31:     socket = io.connect('http://localhost:3000');
32:     $('#notification').hide();
33:     main_loop(socket);
34:   }
35:   catch (error)
36:   {
37:     console.log('Connexion au serveur impossible');
38:     notify('Tentative de connexion au serveur (' +
39:           connect_server.cpt++ + ')...<br /><br />' +
40:           '<progress max="100"></progress>');
41:     setTimeout(reconnect_server, 5000);
42:   }
43: };
...
253: $(document).ready(function()
254: {
255:   /*** Connexion au serveur
256:   connect_server.cpt = 1;
257:   connect_server();
258: }
259: );

```

Au chargement de la page, nous initialisons le nombre de tentatives de connexion (ligne 256) et nous appelons la fonction `connect_server()` définie dans les lignes 25 à 43. Un bloc `try {...} catch {...}` va nous permettre de gérer une connexion impossible. La tentative de connexion se fait en ligne 31. Si elle réussie, alors nous masquons la fenêtre de notification qui a peut-être été affichée suite à une erreur précédente et nous lançons la fonction `main_loop()` qui gère le jeu.

Si la connexion échoue, alors ce sont les lignes 36 à 42 qui sont exécutées. Nous affichons la fenêtre de notification avec un message indiquant l'impossibilité de se connecter et une prochaine tentative (lignes 38 à 39), ainsi qu'une barre de progression infinie (ligne 40). Cette barre, créée par la balise `<progress>` est une nouveauté de HTML 5. Nous relançons ensuite une tentative de connexion au bout de cinq secondes grâce à l'instruction `setTimeout()` de la ligne 41. Cette reconnexion s'effectue depuis la fonction `reconnect_server()` des lignes 19 à 23. En effet, un appel à `connect_server()` ne suffit pas, car si le serveur était arrêté, la ligne HTML de chargement de la librairie `socket.io.js` n'a pas fonctionné ! Il faut donc également charger ce fichier et c'est ce que fait la ligne 21.

Une fois connectés, nous pouvons définir l'emplacement des sprites audio :

```

175:   /*** Définition des sprites de son
176:   var sound = document.getElementById('game_sounds');
177:   var sound_sprites = {
178:     'drag':  [ 0, 0.2 ],
179:     'drop':  [ 1, 1.7 ],
180:     'winner': [ 3, 11 ],
181:     'looser': [ 12, 13 ]
182:   };

```

La boucle principale permet dans un premier temps d'activer les différents éléments déplaçables :

```

106: function main_loop(socket)
107: {
...
185:   /*** On rend les jetons déplaçables
186:   for (var i=1; i<=5; i++)
187:   {
188:     $('#t_' + i).attr('draggable', 'true');
189:     $('#t_' + i).addClass('draggable');
190:   };
191:
192:   /*** On active le déplacement des jetons
193:   // Si la propriété dataTransfer n'est pas ajoutée,
impossible d'y accéder
194:   $.event.props.push('dataTransfer');
195:
196:   $('#tokens').bind({
197:     dragstart: function (event)
198:     {
199:       console.log('Début de déplacement de ' +
200:                   $(event.target).attr('id'));
201:       sound_play(sound, sound_sprites['drag']);
202:       event.dataTransfer.effectAllowed = 'move';
203:       event.dataTransfer.setData("Text",
204:                                   $(event.target).attr('id'));
205:     }
206:   }
207: );
208:
209: // Déclaration des zones de largage
210: for (var i=1; i<=3; i++)
211: {
212:   for (var j=1; j<=3; j++)
213:   {
214:     $('#c' + i + j).bind({
215:       dragover: function (event)
216:       {
217:         console.log('Passage au dessus de ' +
218:                     $(event.target).attr('id'));
219:         $(event.target).addClass('overCase');
220:         return false;
221:       },
222:       dragleave: function (event)
223:       {
224:         $(event.target).removeClass('overCase');
225:         return false;
226:       },
227:       drop: function (event)
228:       {
229:         var elt = event.dataTransfer.getData('Text');
230:         var cible = $(event.target).attr('id');
231:
232:         console.log('Largage de ' + elt + ' sur ' +
+ cible);
233:         sound_play(sound, sound_sprites['drop']);
234:         $(event.target).append($('#' + elt));

```

```

235:     event.stopPropagation();
236:
237:     // On supprime l'effet de pointeur
238:     $('#' + elt).removeClass('draggable');
239:     $(event.target).removeClass('overCase');
240:
241:     // On envoie au serveur la position du pion
joué
242:     socket.emit('played', { col: getCol(cible),
243:                           row: getRow(cible)
244:                         });
245:
246:     return false;
247:   }
248: );
249: }
250: }
```

Nous activons également le déplacement des pions en leur assignant l'attribut `draggable` avec la valeur `true` (ligne 188) et en activant l'événement `dragstart` (lignes 197 à 205). Notez la présence de la ligne 194 activant la propriété `dataTransfer`. Sans cette ligne, le code ne fonctionnera pas avec jQuery (mais on peut toujours appliquer la méthode vue dans l'article sur les contrôles).

La fonction permettant de jouer un son, nommée `sound_play()` et appelée en ligne 201 est la même que celle utilisée dans l'article « Un peu de musique dans ce monde de brutes ».

Il faut également déclarer les zones sur lesquelles les pions pourront être déposés. Chaque cellule `#c..` se voit donc attribuer des événements `dragover` (survol de la case), `dragleave` (fin de survol de la case) et `drop` (« largage » du pion). Les événements `dragover` (lignes 215 à 221) et `dragleave` (lignes 222 à 226) servent à changer la couleur de la case au survol en lui affectant/retirant la classe CSS `overCase`. L'événement `drop` permet d'envoyer un message au serveur avec les coordonnées de la case choisie par le joueur (lignes 242 et 243).

La sauvegarde du score intervient lors de la réception du message `endgame` :

```

132: socket.on('endgame', function (data)
133: {
134:   var bestScore = isBestScore(data.score);
135:
136:   console.log('Fin de partie');
137:   if (data.status == 0)
138:   {
139:     console.log('Egalité!');
140:     notify('Egalité... recommencez !<br />' + bestScore);
141:   }
142:   else if (data.status == 1)
143:   {
144:     console.log('Gagné!');
```

```

145:   notify('Bravo, vous avez gagné !<br />' + bestScore);
146:   if ($('#name'))
147:   {
148:     $('#name').focus();
149:   }
150:   sound_play(sound, sound_sprites['winner']);
151:
152: } else
153: {
154:   console.log('Perdu!');
155:   notify('Ooops... perdu !<br />' + bestScore);
156:   if ($('#name'))
157:   {
158:     $('#name').focus();
159:   }
160:   sound_play(sound, sound_sprites['looser']);
161:
162:
163: /*** On rend les jetons non déplaçables
164: for (var i=1; i<=5; i++)
165: {
166:   $('#t_' + i).removeClass('draggable');
167: }
168: $('#tokens').unbind('dragstart');
169:
170: // Déconnexion du serveur
171: socket.disconnect();
172: }
173: );
```

Les jetons restant sont « fixés » au plateau de jeu par la ligne 168 retirant le lien avec l'événement `dragstart`. La vérification, permettant de savoir si le score du joueur est le meilleur score, est effectuée par la fonction `isBestScore()` appelée en ligne 134. Cette fonction renvoie une chaîne de caractères qui sera affichée dans la fenêtre de notification (lignes 140, 145, et 155). C'est elle qui utilise l'objet `localStorage` pour sauvegarder les données :

```

61: function saveScore()
62: {
63:   var save = {
64:     score: $('#score').val(),
65:     name: $('#name').val()
66:   };
67:
68:   localStorage['bestScore'] = JSON.stringify(save);
69:   $('#theScore').html('');
70:
71:   return false;
72: }
73:
74: function isBestScore(score)
75: {
76:   var getName = '<span id="theScore">' +
77:     '<form onsubmit="return saveScore();">' +
78:     'Votre nom : <input type="text" id="name" /><br />' +
79:     '<input type="hidden" id="score" value="' + score + '" />' +
80:     '<input type="submit" value="Ok" />' +
```

```

81:           '</form>' +
82:           '</span>';
83:
84:     if (localStorage['bestScore'])
85:     {
86:       var bestScore = JSON.parse(localStorage['bestScore']);
87:
88:       if (score < bestScore.score)
89:       {
90:         return 'Vous avez réalisé le meilleur score (' + score + ') !<br />' +
91:               getName;
92:       }
93:     else
94:     {
95:       return 'Meilleur score : ' + bestScore.name + ' (' + bestScore.score +
96:             ')';
97:     }
98:   }
99: else
100: {
101:   return 'Vous avez réalisé le meilleur score (' + score + ') !<br />' +
102:     getName;
103: }
104: };

```

Si le joueur doit saisir son nom, le code HTML est créé dans les lignes 76 à 82, puis inséré dans la chaîne de caractères renvoyée par la fonction. Il s'agit d'un micro-formulaire appelant la fonction `saveScore()` des lignes 61 à 72 lors de la validation. Cette fonction crée un objet contenant le nom et le score du joueur, puis après l'avoir sérialisé, elle l'enregistre (ligne 68). C'est pour cela que dans la fonction `isBestScore()`, pour pouvoir utiliser cette donnée sauvegardée, il faut auparavant la dé-sérialiser (ligne 86).

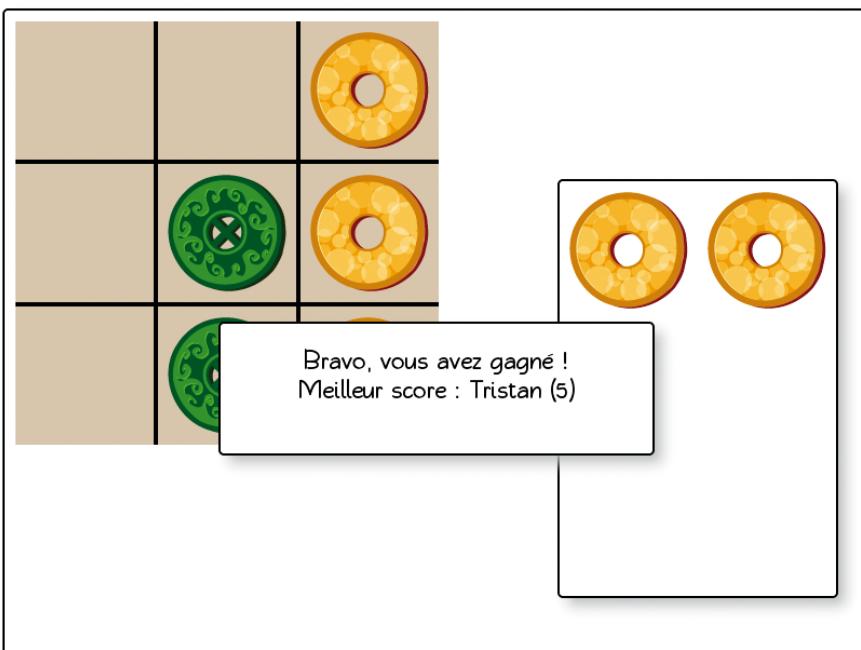


Fig. 2 : Notre jeu de Tic-tac-toe en action

7. Conclusion

En utilisant différentes techniques vues tout au long de ce hors-série, nous avons pu créer un jeu de plateau simple pouvant évoluer vers le mode multijoueur (voir figure 2). À partir de ce code, nous disposons du squelette permettant de réaliser n'importe quel jeu de plateau possédant des règles forcément plus compliquées (il serait difficile de faire plus simple que le morpion...). La tâche la plus complexe sera alors le développement de l'intelligence artificielle permettant à la machine de jouer correctement.

Dans ce type de jeu, l'algorithme le plus couramment utilisé est le min-max avec élagage alpha-bêta [4][5], permettant de choisir le niveau de l'ordinateur. En effet, l'ordinateur calcule un arbre représentant l'ensemble des coups possibles (pour lui et son adversaire) et choisit le coup lui donnant le meilleur score, le score étant calculé par une fonction d'évaluation. En restreignant la profondeur de calcul de l'arbre, on restreint l'« intelligence » de l'ordinateur... Allez, au travail ! ■

Références

- [1] Site officiel de l'outil de prototypage graphique Pencil :
<http://pencil.evolus.vn/en-US/Home.aspx>
- [2] Site officiel de LESS :
<http://www.lesscss.org>
- [3] Site de l'extension vim-less pour Vim :
<https://github.com/groenewege/vim-less>
- [4] Description de l'algorithme min-max dans Wikipédia :
http://fr.wikipedia.org/wiki/Algorithme_minimax
- [5] Description de l'algorithme min-max avec élagage alpha-bêta dans Wikipédia :
http://fr.wikipedia.org/wiki/Élagage_alpha-beta



Hébergement de sites web professionnels et infogérance



Course automobile ou entreprise, les bons réglages font la différence.

- **Serveurs Cluster** Haute Disponibilité
- **Serveurs Dédiés**
- **Serveurs Virtuels**
- **Hébergement Mutualisé**
- **Infogérance**

Chaque voiture est différente, chaque circuit a ses spécificités... Pour obtenir les meilleurs résultats une solution d'hébergement doit elle aussi tenir compte de son utilisation.

Avec Nerim, vous disposez d'une équipe d'experts en technologies de télécommunication. Proche de vous et toujours disponible, Nerim vous proposera **une solution d'hébergement pour votre site web, performante, fiable et adaptée aux besoins réels de votre entreprise.**

Et parce que nous partageons les mêmes valeurs, les mêmes besoins et les mêmes envies, Nerim est partenaire de la Pescarolo Team en 2012.

20000 entreprises avancent déjà avec Nerim. Depuis 12 ans nous sommes à leurs côtés pour qu'ils prennent la pole position dans leurs domaines. À votre tour ?

Pour en savoir plus : **09 73 87 00 00 - www.nerim.fr**



CRÉER UN JEU COMPLET, avec carte isométrique

par Sébastien Chazallet

En complément des outils qui ont précédemment été présentés, cet article va montrer la marche à suivre pour créer un petit jeu très simple et assez complet. Au programme : une carte isométrique, des objectifs et des ennemis.



1. Vue isométrique

Lorsque l'on crée un tel jeu, il faut dessiner des tuiles pour concevoir la carte ainsi que les différents objectifs, obstacles, piéges, ennemis ou personnages. Pour ce faire, rien de mieux que le logiciel Inkscape. La première étape consiste à créer une grille isométrique et la seconde, à lire le numéro spécial du hors-série de *Linux Essentiel* qui y est consacré (voir <http://www.ed-diamond.com/produit.php?ref=lpehs2>).

Une fois les tuiles dessinées, la difficulté réside en l'assemblage des différentes pièces. Il faut faire appel à de petites notions mathématiques. Procédons par étapes :

- Chargeons les images :

```
01 :// Définition des images composant le plateau de jeu
01 :var tuiles_filenames = Array("eau.png","desert.
png","prairie.png","montagne.png");
01 :
01 :// Chargement effectif des images
01 :var tuiles_images = new Array();
01 :for (var i=0;i<tuiles_filenames.length;i++) {
01 :    tuiles_images[i] = new Image();
01 :    tuiles_images[i].src = tuiles_filenames[i];
01 :}
```

- Définissons le plateau de jeu :

```
01 :// Définition du plateau de jeu
02 :var plateau = Array(
03 :    [0,0,1,2,2,2,2,2,3,3,3,3],
04 :    [0,0,1,2,2,2,2,2,2,3,3,3],
05 :    [0,0,1,1,2,2,2,2,2,0,3,3,3],
06 :    [0,0,1,1,2,2,2,2,0,0,3,3],
07 :    [0,0,0,1,1,2,2,2,0,2,3,3],
08 :    [0,0,0,0,1,1,2,2,2,2,3,3],
09 :    [0,0,0,0,1,1,2,2,2,2,2,3],
10 :    [0,0,0,0,0,0,0,0,0,0,0,3],
```

```
11 :    [0,0,0,0,0,1,1,2,2,3,0,3],
12 :    [0,0,0,1,1,1,2,2,2,3,0,0],
13 :    [0,0,0,1,1,1,1,1,2,2,3,3],
14 :    [0,0,0,1,1,0,1,1,1,2,3,3]
15 :);
```

- Déterminons certaines constantes :

```
01 :// Largeur et hauteur d'une tuile
02 :var H = 60;
03 :var W = 36;
04 :
05 :// Point de départ du plateau
06 :var X = 40;
07 :var Y = 240;
```

- Dernier détail technique, récupérer l'objet qui nous permettra de dessiner :

```
01 :var canvas = document.getElementById("plateau");
02 :var contexte = canvas.getContext("2d");
```

- Voici l'élément HTML associé :

```
01 :<canvas id="plateau" width="800" height="600"></canvas>
```

- Voici la dernière étape :

```
01 :// Dessin le plateau de jeu
02 :for (i=0;i<plateau.length;i++) {
03 :    for (j=0;j<plateau[i].length;j++) {
04 :        contexte.drawImage(
05 :            tuiles_images[plateau[i][j]],
06 :            (i+j)*H/2+X,
07 :            (i-j)*W/2+Y
08 :        );
09 :    }
10 :}
```

C'est dans le calcul des coordonnées des tuiles que se trouve la plus grande difficulté.

Une fois que le plateau de jeu est conçu, il faut y placer tous les éléments. Ceux-ci étant plus petits qu'une tuile, il faut introduire un décalage pour les dessiner ; soit un élément à placer étant à la position suivante (il s'agit du personnage) :

```
01 :// Position initiale du personnage sur le plateau de jeu
(début à 0)
02 :Px = 4;
03 :Py = 6;
```

Voici le décalage à utiliser pour notre personnage :

```
01 :// Décalage entre le personnage et la tuile / les
objectifs et la tuile
02 :var Poffx = 22;
03 :var Poffy = -2;
```

Voici la différence avec ce qui précède :

```
01 :// Placement du personnage sur le plateau de jeu
02 :contexte.drawImage(
03 :    perso_images[phase],
04 :    (Px+Py)*H/2+X+Poffx,
05 :    (Px-Py)*W/2+Y+Poffy
06 :);
```

2. Scénario

Écrire un scénario consiste à décrire les acteurs (personnage contrôlé par le joueur, ennemis et adjoints – qui aident le joueur - contrôlés par une IA plus ou moins élaborée) et à déterminer les actions qui peuvent être réalisées, celles qui entraînent le gain de la partie et celles qui entraînent sa perte.

Dans notre cas, le joueur est une petite pyramide qui ne peut évoluer que sur herbe. Elle doit grandir pour évoluer sur le sable, puis sur l'eau et ainsi traverser une rivière pour atteindre la sortie. Un ennemi se déplace aléatoirement et peut se déplacer exactement aux mêmes endroits que le joueur. Des pièges sont également posés un peu partout.

L'important est d'avoir les idées claires et de définir précisément ce scénario de manière à bien le décrire dans le code. Dans notre cas, nous avons 3 phases qui offrent chacune des possibilités de déplacement différentes et qui permettent d'afficher ou non certains détails.

Une fois ce scénario établi, il ne manque plus qu'à écrire le code permettant de jouer. D'abord, permettre le contrôle du personnage par le joueur à l'aide d'une fonction spécifique :

```
01 :function ecouter(event) {
02 :    if (paused != 0) {
03 :        return ;
04 :    }
05 :    var keyCode;
06 :    if(event == null) {
07 :        keyCode = window.event.keyCode;
```

```
08 :    } else {
09 :        keyCode = event.keyCode;
10 :    }
11 :    var redessiner = false;
12 :    switch(keyCode) {
13 :        case 38: // TOUCHE HAUT
14 :            if (Px>0 && est_valide(plateau[Px-1][Py])){
15 :                Px--;
16 :                redessiner = true;
17 :            }
18 :            break;
19 :        case 40: // TOUCHE BAS
20 :            if (Px<11 && est_valide(plateau[Px+1][Py])){
21 :                Px++;
22 :                redessiner = true;
23 :            }
24 :            break;
25 :        case 37: // TOUCHE GAUCHE
26 :            if (Py>0 && est_valide(plateau[Px][Py-1])){
27 :                Py--;
28 :                redessiner = true;
29 :            }
30 :            break;
31 :        case 39: // TOUCHE DROITE
32 :            if (Py<11 && est_valide(plateau[Px][Py+1])){
33 :                Py++;
34 :                redessiner = true;
35 :            }
36 :            break;
37 :        case 13: // TOUCHE ENTREE
38 :        case 32: // TOUCHE ESPACE
39 :            if (Px == objectifs_positions[phase][0] && Py == objectifs_positions[phase][1]) {
40 :                phase++;
41 :                redessiner = true;
42 :            }
43 :            if (phase == 3) {
44 :                paused = 1;
45 :            }
46 :            break;
47 :        default:
48 :            break;
49 :    }
50 :    if ((Px == Ex && Py == Ey) || pieges[Px][Py] != 0) {
51 :        gameover = 1;
52 :        paused = 1;
53 :    }
54 :    if (redessiner) {
55 :        dessiner();
56 :    }
57 :}
```

Cette fonction prend en compte la possibilité de mettre le jeu en pause (lignes 2 à 4) et permet de gérer la fin de jeu que ce soit par le gain (lignes 43 à 45) ou la perte de la partie (lignes 50 à 53). Enfin, on ne fait l'effort de redessiner le plateau que lorsque nécessaire (lignes 16, 22, 28, 34, 41 et 54 à 56).

Ensute, il faut écrire une fonction permettant de gérer les mouvements de l'ennemi :

```

01 :function bouger(event) {
02 : if (paused != 0) {
03 :   return ;
04 : }
05 : if (intervale > (intervales - phase)) {
06 :   intervale = 0;
07 :   var axe;
08 :   var badchoice = true;
09 :   while (badchoice) {
10 :     axe = Math.floor(Math.random()*4);
11 :     if (axe == 0 && Ex>0 && est_valide(plateau[Ex-1][Ey])){
12 :       Ex--;
13 :       badchoice = false;
14 :     } else if (axe == 1 && Ex<11 && est_valide(plateau[Ex+1][Ey])){
15 :       Ex++;
16 :       badchoice = false;
17 :     } else if (axe == 2 && Ey>0 && est_valide(plateau[Ex][Ey-1])){
18 :       Ey--;
19 :       badchoice = false;
20 :     } else if (axe == 3 && Ey<11 && est_valide(plateau[Ex][Ey+1])){
21 :       Ey++;
22 :       badchoice = false;
23 :     }
24 :   }
25 :   intervale++;
26 :   if ((Px == Ex && Py == Ey) || pieges[Px][Py] != 0) {
27 :     gameover = 1;
28 :     paused = 1;
29 :   }
30 : }
31 :
32 : dessiner();
33 :}
```

Cette fonction est plus ou moins la même que la précédente (gestion de la pause lignes 2 à 4 et de la perte de la partie lignes 27 à 30), sauf que le mouvement est choisi aléatoirement (ligne 10) et qu'il doit forcément être possible (on recommence donc le choix tant qu'il n'est pas valide (lignes 9 à 24), sans quoi l'ennemi passerait son tour en quelque sorte).

Il ne manque plus qu'à utiliser ces fonctions et les relier à des événements, ce qui se fait dans l'étape d'initialisation par ces deux lignes :

```

01 :document.onkeydown = ecouter;
02 :interval_id = window.setInterval(bouger, 200);
```

3. Petites particularités

Voici le code spécifique permettant de gérer la mise en pause du jeu ou sa réinitialisation, ainsi que l'affichage d'une fenêtre modale présentant la licence. D'abord, quelques liens HTML :

```

01 :<ul id="dropdown" class="mega-menu">
02 :   <li><a href="#game">Jeu</a>
03 :   <ul>
04 :     <li><a href="javascript: pausegame()">Pause</a></li>
```

```

05 :     <li><a href="javascript: initvars()">Recommencer</a></li>
06 :   </ul>
07 : </li>
08 : <li><a href="#us">À propos</a>
09 :   <ul>
10 :     <li><a href="javascript: licence()">Licence</a></li>
11 :   </ul>
12 : </li>
13 :</ul>
```

Il s'agit de liens vers de simples fonctions JavaScript. La fonction **initvars** (ligne 5) est simplement une fonction qui réinitialise les positions de toutes les pièces du plateau de jeu. La fonction **pausegame** (ligne 4) permet de mettre le jeu en pause ou de le reprendre si l'on n'a pas perdu ou gagné :

```

01 :function pausegame() {
02 :   if (paused == 0) {
03 :     paused = 1;
04 :   } else {
05 :     if (gameover == 0 && gamewin == 0) {
06 :       paused = 0;
07 :     }
08 :   }
09 :}
```

La pause fonctionne grâce au fait que les méthodes permettant des mouvements sur le plateau sont bloquées, tel que cela a été vu précédemment. Il s'agit donc simplement de mettre à jour une variable.

La fonction **licence** permet d'afficher la licence dans une fenêtre modale et son originalité est qu'elle la supprime après quelques secondes :

```

01 :function licence() {
02 :   afficher_licence = true;
03 :   old_paused = paused;
04 :   paused = 1;
05 :   dessiner();
06 :   setTimeout(fermer_licence, 2500);
07 : }
08 :
09 :function fermer_licence() {
10 :   afficher_licence = false;
11 :   paused = old_paused;
12 :   dessiner();
13 :}
```

Lorsque l'on affiche la licence, on force la pause du jeu et lorsque la licence disparaît, le jeu reprend son état précédent. La disparition de la fenêtre modale se fait ligne 6, par l'utilisation de la fonction **setTimeout**.

4. Conclusion

Réaliser un jeu en HTML 5 repose en très grande partie sur l'événement **canvas**, qui est une véritable innovation et à l'aide du JavaScript. Grâce à lui, on retrouve des similarités avec le développement de jeu dans des applications, tel que cela se fait à l'aide de bibliothèques comme PyGame pour Python, par exemple.

Un code d'exemple est téléchargeable sur www.inspryration.org. Une démo jouable est également présente. ■

DÉCOUVREZ LA NOUVELLE FORMULE

Actuellement en kiosque !



LINUX ESSENTIEL N°28

- 16 pages supplémentaires
- un nouveau format...
- de nouvelles rubriques : Windows/Linux, Écosystème du Libre, Culture...

DISPONIBLE

CHEZ VOTRE MARCHAND DE JOURNAUX

JUSQU'AU **30 NOVEMBRE** 2012

ET SUR : www.ed-diamond.com



HORS-SÉRIE N°3 DE LINUX ESSENTIEL

le 14 décembre en kiosque !

L'ESSENTIEL POUR BIEN DÉBUTER AVEC GIMP !

Découvrez toutes les nouveautés de **GIMP 2.8**, comment tirer parti des fonctionnalités de base du logiciel, ainsi qu'une série de tutoriels qui vous permettront de retoucher vos photos et de réaliser des créations originales !



À DÉCOUVRIR CHEZ VOTRE MARCHAND DE JOURNAUX DÈS LE **14 DÉCEMBRE** 2012 !

Venez découvrir NOTRE KIOSQUE NUMÉRIQUE !

RENDEZ-VOUS SUR

diamond.izibookstore.com

et retrouvez GNU/Linux Magazine,
MISC, Linux Pratique et Linux
Essentiel en version PDF !!!



Achetez le magazine en PDF et feuillez-le sur votre ordinateur,
votre tablette ou votre smartphone !

Plus d'informations sur : diamond.izibookstore.com

