PROGRAMMEZ!

Le magazine des dévs

SPÉCIAL HIVER 2022

100% JEUX

Comment créer un jeu ?

Les outils Les technologies

Devenir développeur de jeux

Consoles open source

© Matrix

M 01642 - 6H - F: 6,99 € - RD

| UN | NUMÉRO | CODÉ | COMPILÉ | PAR | POUR | LES DÉVELOPPEUSES | ET | LES | DÉVELOPPEURS

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

TOUJOURS DISPONIBLE !

PROGRAMMEZ!

Le magazine des développeurs

N°250
01/02
2022

LES FEMMES ET LA PROGRAMMATION : LANCE-TOI !

- NESTJS
- JAVASCRIPT
- PYTHON
- TEST
- API
- POSTGRESQL
- ACCESSIBILITÉ
- SÉCURITÉ
- GO
- BOT & NO-CODE

photo : © DISNEY - TRON

| UN | NUMÉRO | CODÉ | COMPILÉ
| PAR | POUR | LES DÉVELOPPEUSES
| ET | LES | DÉVELOPPEURS

M 04319 - 250 - F: 6,99 € - RD

Printed in EU - Imprimé en UE - BELGIQUE 7,50 € - Canada 10,55 $ CAN - SUISSE 14,10 FS - DOM Surf 8,10 € - TOM 1100 XPF - MAROC 59 DH

En kiosque, sur abonnement et PDF

www.programmez.com

Contenus

Divers

Programmez! est une publication bimestrielle de Nefer-IT.
Adresse : 57, rue de Gisors 95300 Pontoise – France. Pour nous contacter : redaction@programmez.com

#HS6JEUX

Des jeux et des développeurs : n'est pas GTA qui veut !

Les jeux vidéo ont longtemps fait rêver les développeurs. Beaucoup espéraient rejoindre un studio indépendant ou des géants du secteur. Les places ne sont pas nombreuses et le niveau exigeant. La pression est forte, les délais limites. Depuis 2 ans, le secteur traverse de nombreuses turbulences : nouveaux modèles économiques, nouvelles expériences, mauvaises ambiances et pratiques douteuses, procès, retards multiples.

Un des fiascos les plus spectaculaires de 2020-21 fut Cyberpunk 2017 de CD Projekt. Laborieusement, l'éditeur a tenté de fixer les gros bugs avec des hotfix. Mais les joueurs ont rapidement critiqué la mauvaise qualité du jeu. Il faut dire que le développement de Cyberpunk a été un cauchemar : 7 ans de production, 328 millions $ dépensés ! Le jeu était sans doute trop ambitieux. Et les patchs successifs n'ont pas tout réglé. Un autre jeu d'envergure Ghost Story a déjà 4 ans de retard ! Les grands studios peuvent se permettre de lancer des jeux nécessitant plusieurs années de conception, à condition de réussir à diriger le Titanic numérique. Les retards sont courants. Début janvier, Skull & Bones qui piétine depuis 2017, multiple les problèmes : le co-réalisateur qui quitte le navire en nageant, et un reboot pur et simple d'une partie du code pour intégrer les nouvelles orientations du gameplay !

Sur mobile, les enjeux sont aussi énormes : les gros jeux dépassent 1 milliard de revenus annuels. Les budgets des jeux mobiles sont tabous, car le marché est tellement concurrentiel que les éditeurs évitent d'en parler. Bref, les enjeux sont si énormes que les studios font tout pour sortir les jeux, surtout si les budgets et les développeurs mobilisés sont à la hauteur des ambitions.

Finalement, c'est quoi un jeu vidéo et comment le développer ? Tout au long de ce hors-série hivernal, nous allons parler équipes, technologies, choix techniques, outils, gameplays, assets, modèles économiques. Un « simple » jeu exige d'être développé comme un vrai projet : cahier des charges, spécifications, choix techniques, développements, tests, etc.

Il existe aujourd'hui une multitude de catégories de jeux, différents modèles économiques, différentes manières de consommer le jeu et de les déployer. Les études publiées annuellement par le SELL, bien que centrées sur la France, parlent d'elles-mêmes en montrant un écosystème mouvant d'une année sur l'autre. Demain, ce sera la réalité mixte / augmentée, toujours plus d'IA, le metavers et toujours plus d'immersion. 2021 a été la confirmation du cloud gaming et la guerre ne fait que commencer entre Microsoft, Google, Sony, Facebook, Nvidia, Apple et tout récemment Netflix !

Comme vous le verrez, nous avons volontairement opté pour des technologies plus légères que les références telles que Unreal Engine, Unity ou Godot. Ces environnements et moteurs sont ultra puissants, mais pas simples à coder, nécessitant au préalable une maîtrise du C# ou du C++. Censé éviter l'effort d'apprentissage des langages, le Visual Scripting, qui relève plus de l'argument marketing, a de grosses limites. En vous engageant sur cette voie, l'impasse est tôt ou tard quasiment certaine.

Donc des outils plus légers à la prise en main bien plus rapide, mais surtout qui permettent de se familiariser avec les concepts propres à la création de jeux, communs à tous les outils complexes ou pas. Nous nous abstenons d'utiliser des interfaces qui en décourageraient plus d'un : elles nécessiteraient un apprentissage à part entière avant même de pouvoir produire quoi que ce soit.

=somme(Franck Dubois+François Tonic)
// les dévs de ce numéro

LES PROCHAINS NUMÉROS

HORS SÉRIE #7
PRINTEMPS

Disponible
dès le 20 mai 2022

PROGRAMMEZ!
N°251

Disponible
le 4 mars 2022

INCONTOURNABLE !

Inside <GIT>

100 % CI/CD
100 % GIT

60 PAGES

Disponible sur programmez.com et en impression à la demande sur Amazon.fr

Microsoft + Activision Blizzard : un rachat à 69 milliards

Microsoft a officiellement annoncé le rachat d'Activision Blizzard, l'éditeur à l'origine de Call Of Duty, Warcraft, Overwatch et tant d'autres. Un rachat qui ne passe pas inaperçu, autant pour la place que celui-ci offre à Microsoft dans le secteur du jeu vidéo que pour son montant : 68,7 milliards de dollars, ce qui en fait le plus gros rachat par Microsoft. Loin devant LinkedIn (27 milliards), Nuance (19,7 milliards) ou encore Skype (8,5 milliards). Les jeux vidéo n'ont jamais paru aussi sérieux, et stratégique.

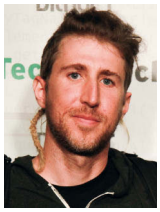
En Ukraine, les bruits de bots se font entendre

Le début d'année en Ukraine a été marqué par une tension géopolitique avec le voisin russe, et celle-ci s'est également fait sentir dans le cyberespace. Plusieurs sites web appartenant à des organisations officielles ukrainiennes ont ainsi été défigurés par des attaques, tandis qu'un logiciel malveillant qui tentait de se faire passer pour un ransomware mais qui ne visait qu'à détruire les données des systèmes touchés a été identifié par Microsoft dans le système informatique de plusieurs clients ukrainiens. Les autorités ukrainiennes n'hésitent pas à pointer la Russie du doigt, mais celle-ci nie toute implication.

Revil : cette fois c'est la bonne ?

Les autorités russes ont annoncé l'arrestation et la mise en examen de huit suspects, soupçonnés d'être les opérateurs derrière le groupe de ransomware Revil. Ce ransomware s'était notamment illustré en s'attaquant au géant de l'agroalimentaire JBS, ou au fournisseur de service Kaseya. Plusieurs suspects étant basés en Russie, la justice américaine demandait à la Russie d'intervenir depuis plusieurs mois déjà. Ce que la Russie a fini par faire, mais rien ne garantit que les suspects soient finalement extradés vers les Etats-Unis.

Goodbye Moxie



Le fondateur de l'application Signal, connu sous le pseudonyme de Moxie Marlinspike, a annoncé qu'il quittait ses fonctions de PDG à la tête de la Signal Foundation, l'organisation qui édite et développe l'application de messagerie sécurisée. Comme il l'a expliqué dans un post de blog, le dirigeant a choisi de quitter son poste pour se tourner vers d'autres aventures et promet que l'application se porte au mieux. Son associé Brian Acton assurera l'intérim en attendant le recrutement d'un nouveau PDG, et Moxie conservera sa place au conseil d'administration de la fondation, afin de garder un œil sur le futur de son bébé.

Exit Hadopi, exit le CSA, bienvenue à l'Arcom

La Hadopi a enfin tiré sa révérence. En tout cas sous sa forme initiale : depuis le 1er janvier 2022, l'autorité de lutte contre le piratage en ligne a officiellement été fusionnée avec le régulateur de l'audiovisuel, le CSA. Les deux entités ont dont donné naissance à l'Arcom, une autorité dirigée par l'ancien président du CSA Roch-Olivier Maistre. Les amateurs du partage de fichier ne devraient néanmoins pas se réjouir trop vite : la nouvelle autorité continuera d'assumer les missions dévolues à la Hadopi dans la lutte contre le piratage, et ambitionne aussi de s'attaquer aux sites de streaming.

Sur npm, un développeur sabote ses propres logiciels

Dans l'écosystème de l'open source, on voit souvent des créateurs se faire pirater. Mais dans certains cas, ils se sabotent eux même. Le développeur à l'origine de deux modules JavaScript, Color.js et Faker.js, a ainsi choisi de publier une mise à jour rendant les deux projets inutilisables. Un moyen pour lui de protester contre l'exploitation de son code par des entreprises fortunées, qui ne reverse rien en échange. Et comme ces deux modules étaient utilisés par un peu plus de 2500 projets pour Faker.js et 19 000 projets pour Color.js, cela a causé pas mal de problèmes. Sans trop se poser de question, Github et npm ont choisi de bannir le développeur contestataire et de rétablir les versions originelles des projets. Et tant pis pour la propriété intellectuelle, business is business.

Qonto, Payfit, Exotec : des licornes à ne plus savoir quoi en faire

Les start-up françaises ont multiplié les annonces de levées de fonds au cours des derniers mois, si bien que le gouvernement se félicite de compter aujourd'hui pas moins de 25 "licornes". Parmi les heureuses élues, on peut citer Payfit, Exotec, Qonto, Lydia ou encore Sorare. Le terme "licorne" désigne les startup non cotées en bourse dont la valorisation est estimée à plus d'un milliard de dollars. Une distinction qui ne signifie finalement rien de plus que l'intérêt des investisseurs en capital risque, mais dont la startup nation se réjouit à grand bruit.

Préparez à vous souvenir du Minitel

Le 30 juin 2012, Orange débranchait définitivement le Minitel, après 30 ans de gloire et de lente décadence. On oublie souvent que l'invention français résista longtemps à Internet et aux nouveaux usages. Programmez! ne manquera pas de parler du réseau des réseaux français, avant Internet.

Les partenaires 2022 de

PROGRAMMEZ!

Le magazine des développeurs

Niveau maître Jedi

softeluent

LA MANUFACTURE CACD2

Niveau padawan

OSAXiS

Vous voulez soutenir activement Programmez! ?
Devenir partenaires de nos dossiers en ligne et de nos événements ?

Contactez-nous dès maintenant :
ftonic@programmez.com

mars

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
			DevCon			
21	22	23	24	25	26	27
			Cloud Sud			
28	29	30	31			

avril

Lun.	Mar.	Mer.	jeu.	Ven.	Sam.	Dim.
				1	2	3
4	5	6	7	8	9	10
			SymfonyLive Paris			
11	12	13	14	15	16	17
			Conférence Metavers			
18	19	20	21	22	23	24
		Devoxx France				
25	26	27	28	29	30	
Android Maker						

GTC : 21 au 24 mars

NVIDIA propose sa grande conférence technique annuelle : la GTC. Des dizaines de sessions, d'ateliers seront jouées. C'est aussi l'occasion pour le constructeur de dévoiler les dernières nouveautés et les roadmaps produits.

Les événements Programmez!

Meetups Programmez!

Les dates du 1er semestre :
8 mars : Design System, REX
5 avril : à venir
10 mai : à venir
28 juin : à venir

Où : Devoteam 43 bd Barbès, Paris
Métros : Château Rouge (ligne 4)
A partir de 18h30

DevCon Serverless + Infrastructure as Code + Kubernes

17 mars

Où : Campus de l'école 42
96 bd Bessières, Paris
Transport : Porte de Clichy (REC C, ligne 13, ligne 14)
T3b : arrêt Honoré de Balzac
A partir de 13h30

Conférence Metavers

14 avril

Où : Campus de l'école ESGI
242 rue du Faubourg Saint-Antoine, Paris
Transport : Nation (RER A, Métro 1, 2, 6, 9)
A partir de 13h30

Informations & inscription :
programmez.com

A VENIR

- AFUP Day 2022 : 20 mai / Lille
- MixIT : 24-25 mai / Lyon
- AlpesCraft : 9-10 juin / Grenoble
- DevFest Lille : 10 juin / Lille
- JFTL 2022 : 13 & 14 juin
- France API : 14 juin / Paris
- Serverless Day : 23 juin / Paris
- Socrates : 24 juin / Rennes
- Hack in Paris : 27 juin – 1er juillet / Paris
- SunnyTech : 30 juin & 1er juillet / Montpellier
- Volcamp : 13-14 octobre / Clermont Ferrand
- DevFest Nantes : 20-21 octobre / Nantes
- Codeurs en Seine : 17 novembre / Rouen

Merci à Aurélie Vache pour la liste 2021, consultable sur son GitHub : https://github.com/scraly/developers-conferences-agenda/blob/master/README.md

LES PROCHAINS NUMÉROS

HORS SÉRIE #7 PRINTEMPS

Disponible dès le 20 mai 2022

PROGRAMMEZ! N°251

Disponible le 4 mars 2022

Le crowdfunding dans le jeu vidéo

Le crowdfunding a dépassé, en France, le montant historique du milliard d'euros collectés en 2020. Il s'agit d'une augmentation de 62% par rapport à 2019 et d'une multiplication par 6 depuis 2015. Kickstarter a été un des premiers. En France, la plateforme Ulule est la plus connue.

Jean-Charles IZE

Jean-Charles a cofondé Mens Sana Games pour porter des projets ludiques et de gamification. Mens Sana Games a pu voir le jour en grande partie grâce au crowdfunding.

Ce moyen de financement est utilisé dans de nombreux secteurs d'activité. Dans l'industrie du jeu, il s'agit d'une méthode bien connue. Il est d'ailleurs probable que vous, lecteur, lorsque vous entendez le terme de *crowdfunding*, pensiez au financement participatif d'un produit ludique ou gaming plutôt qu'à la construction d'un immeuble ou à l'investissement dans une entreprise. Mais dans l'immobilier aussi, le crowdfunding existe. Et certaines entreprises développent leur actionnariat de cette manière. **Figure 1**

Les origines du *crowdfunding*

Le succès des plateformes en ligne de financement participatif a contribué à l'essor du *crowdfunding*. Mais ce moyen de financement de projets est bien plus ancien qu'internet. De nombreux projets à travers l'histoire ont été financés par sollicitation de la population. Tout au long de l'Ancien Régime, des initiatives en matière de charité partaient de la sollicitation de la population.

Les chevaliers fondateurs de l'Ordre des Templiers ont réalisé une véritable levée de fonds auprès de multiples seigneurs à travers l'Europe pour réaliser leurs projets.

En 1875, la Statue de la Liberté a aussi été financée par souscription publique de près de 100 000 Français.

En 1882, c'est la Sagrada Familia à Barcelone qui a été *crowfunded* dans une campagne qui est probablement la première connue en matière immobilière.

Aujourd'hui, depuis MyMajorCompany (1re plateforme de financement participatif ayant connu le succès), le *crowdfunding* est entré dans une autre dimension, celle d'internet. Et de multiples projets ont pu voir le jour grâce à ce moyen de financement, à la portée de tous. Voici mon témoignage.

Comment j'ai utilisé le *crowdfunding* dans l'édition de jeu

En 2020, j'ai lancé ma première campagne pour financer un jeu de société de niche (*Dungeon of Fitness*) que j'ai développé entre 2018 et 2019. Je n'avais auparavant jamais participé à une campagne de *crowdfunding*. Je ne m'y intéressais pas. C'est mon associé, cofondateur de Mens Sana Games (studio créé à l'occasion de la sortie de ce premier jeu), Julien, ludiste passionné, qui était habitué (voire accro :-)) à cette pratique. Il m'a convaincu de l'opportunité de financer ce premier jeu par ce moyen. Ce fut un succès. En réalité, ce fut plus qu'un succès. Ce fut une révélation. Aujourd'hui, moins de 2 ans après notre première campagne, nous en sommes à notre 5e et bientôt 6e campagne (en comptant nos jeux et ceux de nos clients).

Mens Sana Games a été créé pour éditer le jeu *Dungeon of Fitness*, mais il s'est avéré que son succès raisonnable avait attiré l'attention d'entreprises qui recherchaient certaines compétences. Une découverte suite à cette 1re campagne : le *crowdfunding*, au-delà de la collecte de fonds pour financer un projet, est un outil de communication redoutable.

Ce qui était évident, et qui m'a convaincu immédiatement, c'est que le *crowdfunding* est un excellent moyen de lancer un projet en autoédition qui, par ailleurs, comporte peu de risques. Lever des fonds auprès de ses clients est, selon moi, la meilleure méthode d'entreprenariat.

Ce moyen fonctionne, non seulement pour le jeu de société, mais aussi pour le jeu vidéo.

Quelques exemples de jeux vidéo financés en *crowdfunding*

Sur *Ulule* (principale plateforme francophone) :

• *Battle for Egadia*
Jeu de cartes style Magic avec une interface à la *Hearthstone* dans un univers d'animation occulte. 10 000 € recherchés, 40 000 € obtenus.

• *Noob, le jeu vidéo*
RPG old school - combat au tour par tour. 90 000 € recherchés, 1 246 000 € obtenus.

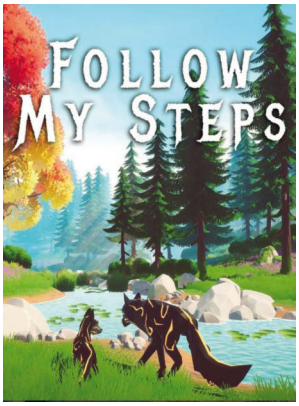
• *Botaki*
App visant à éduquer les enfants concernant la diversité des fruits et légumes alimentaires et à leur apprendre à jardiner. 15 000 € recherchés, 20 000 € obtenus.

Figure 1

Sur *KissKissBankBank* (plateforme francophone appartenant à *La Banque Postale*) :

• *Fairymm*
Horreur - point & click.
15 000 € recherchés, 19 000 € obtenus.

• *Tawö*
Jeu mobile - jeu éthique - jeu éducatif - sidescroller 2D style *Mario*.
22 000 € recherchés, 22 000 € obtenus.



• *Follow My Steps*
Open world aventure, dans la veine *Okami* ou *Breath of the Wild*.
30 000 € recherchés, 38 000 € obtenus.

Des projets comme Star Citizen, Kingdom Come : Délivrance, Elite : Dangerous, Pillars of Eternity, Divinity, Darkest Dungeon et bien d'autres encore ont été financés sur Kickstarter.

Mais comment un projet devient-il un succès ?
Si vous écumez les plateformes pour vous inspirer, vous verrez des projets morts ou en difficulté. Dès lors, suffit-il d'avoir une bonne idée, de l'exprimer sur une plateforme pour en espérer un effet viral immédiat, le fameux effet boule de neige, qui permettrait de la financer avec succès ? Finalement les projets non financés n'étaient peut-être pas de bonnes idées… Ce n'est pas du tout ce que je pense.

Quelle recette pour lancer un *crowdfunding* sur un jeu vidéo ?

Tout d'abord, commençons par les ingrédients :

• Un pitch
Votre projet doit être accrocheur : quelques mots, en 3 phrases. Il faut le travailler, le tester. Ne gardez pas vos projets secrets. Il est improbable de se faire voler son idée. Cela existe, évidemment, mais c'est rare. Et puis quelqu'un l'a probablement eue avant vous. Quand j'ai lancé *Dungeon of Fitness*, je pensais être le premier à mélanger *Heroic Fantasy*, RPG et Sport… je me trompais… mais ce n'est pas grave. N'ayez pas peur, testez vos idées avant même de travailler dessus.

• Un projet
Pour imaginer que quelqu'un vous donne une somme d'argent pour recevoir hypothétiquement une contrepartie, il faudra non seulement avoir bvien exprimé l'idée (le pitch) mais, en plus, il faut donner envie avec un contenu gratuit, disponible et déjà satisfaisant et alléchant. Ce contenu peut être un prototype, une version beta ou encore une démo, qui servira à la communication autour du projet. Vous devez offrir quelque chose à votre communauté, ou future communauté, pour qu'elle vous aide en retour. Mais attention ! Il ne s'agit pas de développer quelque chose de trop complexe. Vous devez ensuite imaginer une MVP (*Minimum Viable Product*) qui sera la première étape à atteindre de votre projet. L'ambition doit être proportionnée et ne doit pas laisser transparaître un gouffre financier. Toutefois vous devez proposer une contrepartie qui démontre l'intérêt de votre travail, pas plus, pas moins.

• Une audience ou une communauté
Une plateforme de *crowdfunding* offre une audience à travers sa propre communauté. Elle est colossale mais plus ou moins qualifiée. Il faut garder en tête que la plateforme présente beaucoup de projets et n'est qu'un outil. Ainsi, il ne vaut mieux pas compter sur l'audience de la plateforme. De fait, forger votre propre communauté est l'assurance d'avoir une adhésion et un intérêt progressifs à votre projet. En visant une niche (passion, sport, hobby…) vous parviendrez à fédérer plus facilement, d'autant plus si aucun jeu n'existe et que votre prototype est accessible sans peine.

• Un plan financier
Pour demander de l'argent, il faut savoir pourquoi. Il faut bien calculer et mesurer. Cela revient à faire un *business plan*. Vous n'êtes pas obligé de dévoiler ce plan. Je n'y suis pas favorable. La plupart des plans communiqués lors des *crowdfunding* sont probablement faux, et puis « aucun plan ne survit au contact de l'ennemi ». Mais il faut avoir un plan, ne serait-ce que pour comprendre et maîtriser votre projet au-delà de la version MVP afin d'en définir les contours. Le plan doit intégrer tous les scenarii, les bons et les mauvais. Si vous faites un excellent score, il faut que les bonus débloqués soient aussi bien calculés que votre MVP. Il faudra peut-être également envisager des coûts cachés et dans le cas d'une campagne un peu serrée, il faudra envisager de mobiliser sa trésorerie personnelle.

• Un plan de communication
Une fois votre plan financier établi, vous devez planifier votre communication sur 20, 30 ou même 60 jours. Une campagne de *crowdfunding* est très intense ! Vous devrez être inventif pour préparer le lancement, qui est le moment le plus crucial, serrer les dents quand il ne se passera pas comme prévu et, enfin, être endurant quand la campagne ralentira en milieu de période avant de s'accélérer sur la fin. C'est comme une bonne course, il faut un bon démarrage, tenir le

rythme tout au long de la campagne et être capable d'accélérer à la fin. Personnellement, j'aime nourrir la campagne tout au long de la période avec des partenariats de communication qui stimulent les algorithmes des plateformes de *crowdfunding*. Et oui, le référencement au sein même de la plateforme est crucial pour profiter au maximum de son audience.

Voici maintenant la recette :

Parlez de votre projet, testez-le auprès de votre cible le plus vite possible. Ne passez pas des années à développer (avec passion certes) le jeu de votre vie dans votre cave. Personne n'y jouera. Commencez simple et petit. Les premiers joueurs vous guideront et vous stimuleront pour mieux développer. Mais, surtout, votre idée va mûrir, tout autant que le pitch qui l'accompagne. Vous aurez également forgé la meilleure MVP qui soit, celle qui est jouée, débuggée et appréciée.

Vous pourriez vous arrêter là… Mais vous voulez allez plus loin. Après tout, si vous avez créé, c'est pour que cela est joué et en tant que créateur je peux comprendre, on ressent beaucoup de gratification à avoir diverti quelqu'un avec succès.

Vous devez définir vos besoins : un revenu nécessaire pour couvrir le temps de développement, un graphiste à payer ? Il faut définir votre *business plan* tout en concevant également le modèle économique de votre jeu. Pas besoin d'imaginer quelque chose de nouveau. Regardez ce qui marche déjà dans le secteur du jeu vidéo. L'avantage principal du *crowdfunding* est que vous financez vos besoins en amont du projet. Mais attention, ne surestimez pas les contreparties que vous offrez. Il est d'usage d'offrir des paliers bonus en fonction du montant de financement. Ne les improvisez pas et protégez-vous. Il ne s'agit pas de sortir de votre budget en proposant X suppléments. Cela ne fera pas venir plus de monde sur votre campagne. Remerciez vos contributeurs avec simplicité, un bonus que vous pouvez vous permettre et une production principale de qualité.

En somme, un bon pitch, un bon prototype, une communauté engagée, un plan financier rigoureux, un plan de communication ciselé… vous atteindrez le succès.

Et enfin, la liste des bêtises à éviter :

- Mal évaluer les frais de port attachés aux objets publicitaires matériels (ou toute autre erreur budgétaire). Ce fut le cas d'un porteur de projet qui, pour livrer ses contributeurs, a choisi de vendre sa maison car il avait oublié de prendre en compte les frais de port sur une campagne mondiale. Sans tomber dans ces extrêmes, il est toutefois judicieux de ne pas oublier certains postes de dépense, pour ne pas transformer brutalement un succès en échec alors même que la campagne était réussie. Faites simple pour ne pas vous tromper !
- Ne pas être transparent sur le déroulement de la campagne et du projet… il faut communiquer ! Certains porteurs de projet rencontrent des problèmes et n'en parlent pas. Ils estiment que ce n'est pas prioritaire, ils ont peut-être honte aussi. En *crowdfunding*, il faut parler à sa communauté et à ses contributeurs. Il faut leur raconter l'histoire de la création de ce qu'ils ont financé. Le retard sera pardonné s'il est raconté et expliqué. Faire une apparition surprise à la date de sortie prévue pour expliquer qu'il y aura 6 mois de retard (comprendre 1 an) est très, très, très mal perçu.
- Surévaluer ou sous-évaluer votre succès. Un « like » sur *Facebook* en amont du projet ne vaudra jamais une vente. Il n'est pas facile de vendre un produit et, à l'heure de l'immédiateté, n'imaginez pas qu'il est facile de vendre un produit avec une livraison à 6 ou 8 mois. Avoir une petite bande de contributeurs c'est déjà énorme, elle constitue une preuve sociale et une preuve de marché. Alors, il faut apprécier les petites victoires, 100 contributeurs c'est énorme. 200 fans sur *Facebook*, quant à eux, ne représenteront peut-être que 10 contributeurs.

Conclusion

Après plusieurs campagnes de *crowdfunding* réussies pour fabriquer des objets matériels. Je travaille sur un prototype de jeu vidéo, gageons qu'il servira de MVP pour une prochaine campagne au cours de laquelle j'appliquerai la même méthode !

Frédéric Larivé
(alias Eviral)
Développeur fullstack à Montpellier
Créateur amateur de jeux vidéo
www.facebook.com/bloodianchronicles
www.bloodianchronicles.com
bloodark@hotmail.fr

Retour d'expérience sur un projet indé atypique
The Bloodian Chronicles

J'avais 13 ans quand L'arche du Captain Blood est sorti en 1988. Ce jeu de science-fiction étrange a marqué mon adolescence, graphismes sublimes de Didier Bouchon, scénario et gameplay enivrant de Philippe Ulrich, musique de Jean-Michel Jarre. Un OVNI, une bombe de créativité, un jeu d'auteur signé par la petite équipe d'ERE informatique/EXXOS (SRAM, Crafton et Xunk, Teenage queen, le passager du temps, Kult, Despotik Design...). Ce jeu est devenu un véritable mythe pour de nombreux retrogamers, une perle artistique et technique, un chef-d'œuvre pour les quarantenaires nostalgiques. Ma vocation de développeur vient sûrement de la magie exercée par les jeux vidéo de mon enfance. ERE Informatique deviendra CRYO Interactive quelques années plus tard, un des fleurons de la french touch de la fin des années 1990. **Figure 1**

En 2009, je retombe sur des screenshots de Captain Blood sur le web et je décide sur un coup de tête de développer une « suite » juste par passion et par nostalgie. Modélisation sous 3ds max, pas mal de Photoshop et une maquette faite sous XNA en C# (RIP). Au bout de quelques mois, j'avais un début de jeu, je décide alors de contacter Philippe Ulrich et Didier Bouchon par mail pour leur présenter mon travail. Je ne m'attendais pas à grand-chose, après la fin de CRYO ils s'étaient retirés du jeu vidéo... À ma grande surprise, ils se sont montrés emballés par ce que j'avais fait !

C'était un rêve de gosse de pouvoir rencontrer et travailler avec ces grands noms du jeu vidéo made in France.

Après plusieurs rencontres sur Paris et des centaines d'heures de Skype, le projet Captain Blood Legacy (suite officielle de Captain Blood 30 ans après) était né. Il fallait trouver des investisseurs pour monter un petit studio et constituer une équipe. Les années passèrent, aucun investisseur téméraire à l'horizon, des tentatives de crowdfunding avortées... 2012 fut la fin de Captain Blood Legacy, projet mort-né, car trop hors piste (bio game MMO). C'est à ce moment que nos routes se séparèrent. **Figure 2**

L'idée

Le coup fut dur à encaisser, mes rêves d'intégrer le monde du jeu vidéo s'envolaient. Tremper dans le monde du jeu vidéo, c'est le rêve de beaucoup de développeurs. Le jeu vidéo combine l'artistique à la technique et permet de véhiculer de l'émotion à travers un monde imaginaire que l'on a créé de ses mains. J'ai décidé de repartir de zéro. Je devais repenser le projet en fonction de la réalité : j'étais seul. The Bloodian Chronicles est un fan game indé, non officiel et gratuit : nouveau gameplay, nouveau scénario, nouveaux personnages. Je suis parti sur un point'n click (Indiana Jones, Monkey Island...) en 2,5D (personnages en 3D et décors en 2D) inspiré par l'univers de Captain Blood. Ce jeu d'aventure SF sera divisé en 2 épisodes de 12 chapitres.

L'avantage d'être seul c'est d'être libre de ses choix et orientations, pas d'argent en jeu, on peut laisser l'intuition nous guider sans rien ni personne pour freiner nos idées. La créativité n'est pas bridée par la peur de l'échec. **Figure 3**

Les objectifs

Avec le recul, il faut quand même bien poser les bases de ce que l'on veut faire avant de coder :

- Pourquoi faire un jeu ? pour le plaisir, l'argent, la gloire ?
- Est-ce faisable si je suis seul et en combien de temps ? (Essayez d'être objectif)
- Est-ce que je suis prêt à y passer de très nombreuses heures un peu tout seul dans mon coin ou avec 2 ou 3 potes ?
- Quel style visuel ? pixel art, cartoon, 2D, full 3D ?
- Quel genre ? Aventure, Shoot 'em up, plateformer, Battle royal, FPS, réflexion...
- Quel public ? Quelles plateformes ?

Figure 3

Figure 1

Figure 2

En fonction de vos réponses, vous n'allez pas vous orienter vers le même style de jeu. Fixez-vous des objectifs atteignables. La route pour créer un jeu (et le finir) est longue, solitaire, ingrate, mais gratifiante. Posez-vous les bonnes questions, anticipez, suivez votre instinct, mais de façon raisonnée.

Le draft

Par où commencer ? Cela dépend du genre de jeu... Pour un jeu d'aventure, c'est par l'histoire, le scénario, les lieux et les personnages qu'il faut commencer. Comme un bon roman ou un bon film, il faut des personnages attachants et une intrigue qui vous tient en haleine avec des cliffhangers bien placés pour vous donner envie de continuer. Il faut que les premières minutes de jeu arrivent à captiver le joueur sinon il passera rapidement à autre chose (onboarding). Votre jeu doit être simple à comprendre, à prendre en main, clair, facile au début pour accrocher le joueur et ne pas le perdre.
J'ai passé 3 semaines à travailler le scénario du jeu, c'est la clé de voûte d'un jeu d'aventure. J'ai imaginé le personnage principal, Loon, un anti-héros puisque c'est une jeune fille de 12 ans amnésique qui se réveille à bord de l'arche, un vaisseau vivant à la fois biologique et mécanique. J'ai ensuite imaginé les lieux et les personnages secondaires. J'ai créé des fiches descriptives et réalisé de nombreux croquis. Comme une bonne série Netflix, il faut une mise en place progressive de l'histoire, des éléments perturbateurs, des péripéties, des rebondissements, des solutions, un dénouement. La fin doit être inattendue et se finir sur un coup de théâtre. J'ai donc commencé mon scénario en partant de la fin. Je l'ai découpé en 24 courts chapitres, le tout scindé en 2 épisodes (2 délivrables). Au début on fait des choix puis on les adapte, rien n'est figé, tout est en mouvement perpétuel durant la conception. **Figure 4**

Le POC

Commencez par un POC pour valider votre concept. Les premiers joueurs vous feront des retours très utiles. Personne n'a la science infuse, on pense avoir raison, mais la réalité du terrain l'emporte.
Le POC jouable doit clairement montrer le gameplay, c'est le cœur du jeu. On évite le mode tunnel qui consiste à finir le jeu et à la fin se rendre compte qu'il n'intéresse personne. Vous pouvez à la rigueur coder salement pour le POC, récupérer des assets graphiques à droite à gauche pour gagner du temps et vous focaliser sur le dev du gameplay. Si le POC est concluant auprès de votre famille, de vos amis et de quelques inconnus alors ça sent bon. Dans le cas contraire, il faut réitérer pour améliorer/corriger ou envisager de changer d'idée. Une fois le POC validé vous pouvez refactorer, faire du code propre, bien architecturer votre projet, l'habiller et faire du polish. Le POC est une étape difficile, mais une fois que c'est passé vous vous sentez le roi du monde et la motivation c'est le nerf de la guerre pour aller jusqu'au bout. La Beta 1 de The Bloodian Chronicles (4 premiers chapitres) a fait office de POC et j'ai traité la plupart des retours que j'ai eu dans la Beta 2. J'attaque plus sereinement les chapitres suivants.

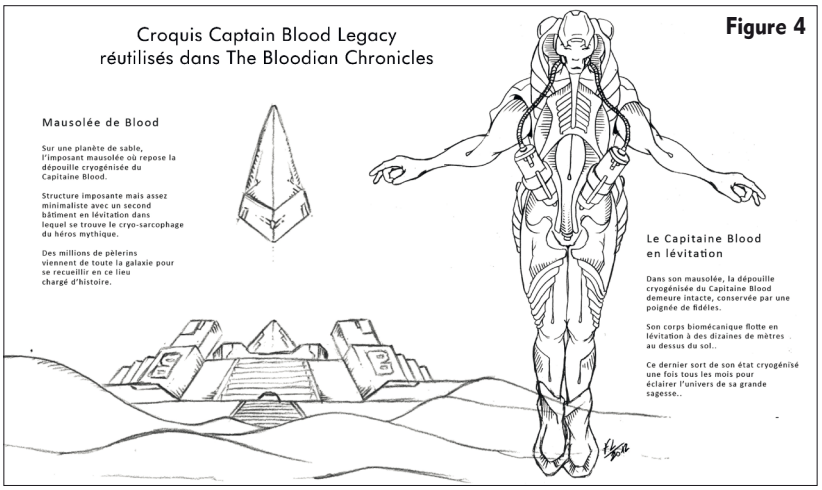


Figure 4

L'équipe et les compétences

Pour développer un jeu il faut un ensemble de compétences. Un dév est rarement un bon graphiste ni même un bon testeur. C'est là où il faut monter une petite équipe si on n'a pas toutes les compétences. On peut apprendre, se former pour combler ses lacunes, mais ça prend du temps.
Par chance, j'ai la double casquette graphiste/développeur ce qui me permet d'être autonome. J'ai commencé par le graphisme en pixel art sur Amstrad CPC (OCP Art Studio) puis sur Amiga 500 (Deluxe Paint) dans les années 90 pour faire du Photoshop et du 3ds max aujourd'hui. Je suis développeur de formation et j'exerce dans ce milieu depuis 20 ans.
Monter une équipe peut être très simple ou très compliqué. Si vous êtes à 2 ou 3 à trouver une idée, c'est l'idéal, car votre motivation est partagée. Sinon, attention à la motivation à géométrie variable ou temporaire des autres membres de l'équipe. Chacun doit trouver sa place et trouver un intérêt au projet (rétribution morale dans les crédits du jeu, pourcentage des futures ventes...). Attention aussi dans le cas d'une équipe à être clair sur qui fait quoi et sur le partage de la propriété intellectuelle. En cas de séparation, à qui appartient le projet ?
Le projet repose sur vos épaules, ayez une motivation en acier pour passer les périodes difficiles.

Le gameplay

Le gameplay est le point le plus important du jeu. Vous pouvez reprendre un gameplay existant, le faire varier en lui apportant un petit plus, en mixer plusieurs entre eux ou carrément inventer le vôtre de A à Z (concept innovant).
Essayez de quantifier le nombre de joueurs que vous allez pouvoir toucher avec tel ou tel gameplay (casual games vs jeux d'auteurs). Le graphisme, les animations, les effets sonores et la musique viendront sublimer votre gameplay s'il est bon. Si ce n'est pas le cas, ils ne sauveront probablement pas votre jeu. **Figure 5**

Le public visé

En fonction du gameplay, vous allez pouvoir cibler un public. Vous avez le choix entre un public très large (hommes, femmes, enfants de tous les âges) et un public plus averti donc plus réduit (gamers, amateurs de SF...). The Bloodian Chronicles est un jeu entre les deux je pense. Le côté SF réduit le public, mais le côté aventure et point'n click peuvent plaire à un public assez large. J'ai essayé de travailler une

ambiance étrange et des concepts SF innovants quitte à perdre du monde plutôt que de faire un jeu trop lisse et trop fade. Equilibre difficile à trouver, mais c'est mon choix, j'espère qu'il est bon.

Les milestones

Dès le début du projet essayez d'imaginer les livrables que vous allez produire. Sortez des versions au fur et à mesure pour les faire tester. Essayez de vous fixer des jalons, mais pas de dates à tenir, car quand on est seul et qu'on fait cela sur son temps libre c'est un peu illusoire et contre-productif. Sauf si des opportunités de montrer votre jeu s'offrent à vous (salons, chaîne YouTube ou Twitch, articles de magazines) et vous imposent d'avoir quelque chose de montrable à une date donnée. Produire un jeu c'est une course de fond donc limitez les freins inutiles.

De mon côté j'ai mis 5 ans pour sortir une Beta1 propre et bien testée avec les fondations du jeu que je peux maintenant réutiliser et gagner du temps. Le début est long, car on part de presque rien.

L'environnement de dev

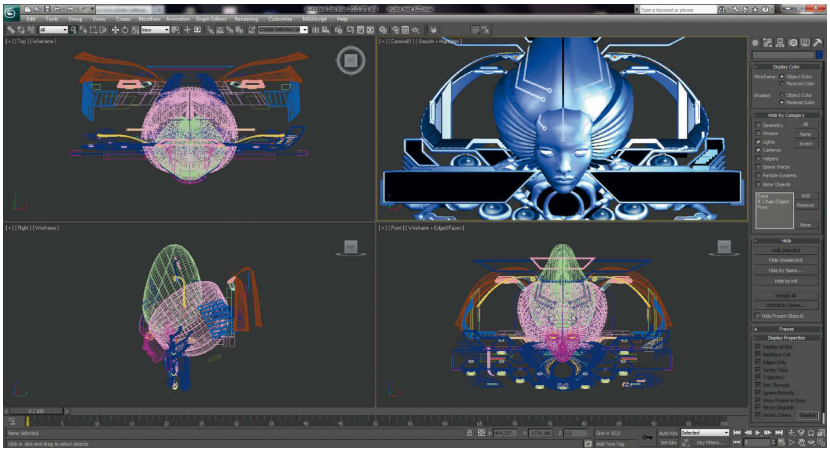
Je développe sur un portable DELL Inspiron grand public sous Windows. Avec un bon SSD, pas mal de RAM et une carte graphique grand public récente vous pourrez développer sans trop de soucis, pas besoin d'avoir une machine de guerre ou du matériel de pro. Le jour où vous ferez un AAA ce sera autre chose, mais pour un "petit" jeu indé un bon portable (ou desktop) suffit. J'utilise VS2022 Community (gratuit) avec Unity 2020 Personal (gratuit), mais j'ai tendance à basculer vers VSCode, car il est plus léger. Pour le graphisme 2D, je suis sous Photoshop CS6, pour la 3D sous 3ds max 2020 et ZBrush 2020.

Pour les effets sonores et la musique, j'utilise Goldwave et Audacity, je retravaille des fichiers audios existants, mais je ne compose rien, je ne suis pas musicien. Mes sources sont sur un repo privé bitbucket (gratuit) et je sauvegarde tous mes gros fichiers sources (PSD Photoshop, fichiers 3ds max...) sur mon OneDrive de 1To (gratuit, car offert avec l'achat de mon mobile Samsung il y a 6 ans !)

Bref, on peut trouver beaucoup de choses gratuites pour démarrer un projet sans dépenser un centime. Parfois il faut bidouiller, contourner, ruser, mais on trouve toujours des moyens de s'en sortir gratuitement. **Figure 6**

Figure 5

Figure 6



L'organisation

Je ne m'impose rien, je suis souple et j'évite de me poser trop de contraintes. Cependant il faut être rigoureux dans ce que l'on fait, mettre la poussière sous le tapis parce qu'on a la flemme de faire une tâche rébarbative ce n'est pas une solution. Ça m'arrive de perdre des heures en bloquant sur quelque chose, sur un détail parfois, mais étant d'un tempérament ultra têtu j'y passe le temps qu'il faut. J'ai parfois été obligé de casser des pans entiers de mon jeu pour les refaire, car après coup je me suis rendu compte que j'avais mal fait. C'est dur sur le moment, mais nécessaire pour la pérennité du projet. Il faut régulièrement refactorer, améliorer, peaufiner, optimiser, tester et retester encore... Je suis scrupuleusement le scénario que j'ai écrit pour développer les nouveaux chapitres, mais je rajoute souvent au feeling des détails, des choses secondaires. J'ai décidé de ne pas écrire de tests auto, c'est discutable, mais le travail est déjà tellement colossal que je m'épargne l'apprentissage des tests auto d'un jeu sous Unity et les multiples cas de tests à écrire. Je reteste manuellement régulièrement mes différents écrans tout au long du dev et étant donné que je travaille seul j'ai relativement peu de régressions, car je sais ce que je touche et les impacts. **Figure 7**

Les plateformes

Étant sous Windows depuis de nombreuses années j'ai tout naturellement commencé le dev sous Windows et ciblé Windows pour mes premiers builds. Je me suis assuré dès le début que mon jeu tournait de la même façon sur Android. Une des forces de Unity c'est d'avoir un code unique que l'on peut builder sur plus de 20 plateformes différentes (Windows, macOS, Linux, Android, IOS, WebGL, consoles...). J'ai fait le choix de sortir le jeu à terme sur Windows, macOS, Android et iOS. 4 plateformes c'est déjà beaucoup, car il y a toujours de petits soucis à régler d'une plateforme à l'autre (différence de comportement avec le tactile, avec les shaders entre DirectX et OpenGL...). En ciblant Windows et macOS, je cible un public de joueur plus âgé (plus gamers, plus old school) alors qu'avec Android et iOS je cible un public plus jeune, plus casual, plus large. Côté distribution, le jeu est en téléchargement libre et gratuit sur itch.io et GameJolt. À terme il sera sur le Windows Store, le Play Store, l'App Store et peut être un jour sur Steam. **Figure 8**

Les choix techniques

Étant développeur C#, j'ai fait le choix de Unity et je ne le regrette pas. Unreal Engine est un autre très bon environnement, mais en C++. Unity et Unreal Engine se valent sur de

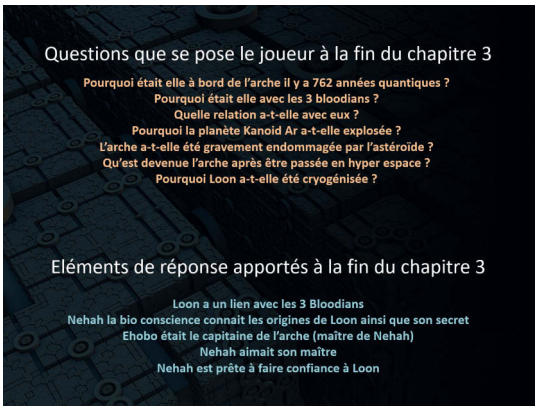


Figure 7

nombreux points, les deux sont ultras complets, vous n'aurez presque aucune limite. Même si l'IDE de Unity est très bien pensé et fait gagner beaucoup de temps, il faut quelques mois pour dompter la bête. Concernant le moteur de dialogues du jeu, je suis parti sur le moteur de chatbot ALICE (open source) créé par le Dr. Richard Wallace que j'ai pas mal étoffé. Restez au maximum en terrain connu, ne multipliez pas inutilement les nouvelles choses à apprendre. **Figure 9**

Les moyens de gagner du temps

Je vous recommande de piocher dans des bibliothèques d'assets existants (images, sprites, modèles 3D, animations, sons, musiques, starter kits, sources Github...). Ce n'est pas une honte utiliser du contenu fait par quelqu'un d'autre, il faut juste respecter la licence, acheter quand c'est payant ou faire une attribution en nommant l'auteur quand la licence l'impose. Certains auteurs sont ravis de voir leur travail intégré dans un jeu et vous autorisent à utiliser gracieusement leur contenu. Quelques sites que j'utilise régulièrement : CGTrader, Turbosquid, Sketchfab, Mixamo, Unity assets store, Noun project, Game Icons, Freesound...

Quand je coince trop sur quelque chose que je n'arrive vraiment pas à faire, je passe parfois par Upwork, un site ou des freelances proposent leurs services. Il y a des pointures dans tous les domaines et c'est vous qui proposez la rémunération pour la tâche à effectuer. Autre conseil, ne multipliez pas trop les projets, concentrez-vous à fond sur un sujet à la fois : votre jeu. L'objectif c'est de le finir !

Le financement

Vous avez le choix entre créer un jeu indé gratuit, un freemium (entre les deux) ou payant. Les investisseurs ou les éditeurs ne s'intéresseront pas à un jeu non commercial puisqu'il ne leur rapportera rien. Vous pouvez tenter le crowdfunding (kickstarter, ulule...) ou les dons (tipeee, patreon...).

J'ai développé en parallèle un outil nommé Unity Optimizer pour les besoins de mon jeu. Il me permet de réduire la taille de mes builds en analysant le projet Unity et en trouvant toutes les optimisations à faire. J'ai une version gratuite (limitée) et une payante (complète). Les revenus des quelques centaines de ventes me permettent d'acheter des assets et de gagner un temps précieux.

Le Marketing

La partie market est cruciale. Si vous ne communiquez pas dessus, personne ne viendra s'intéresser à votre projet.

Le web est un océan, votre projet est une goutte d'eau dans cet océan. J'ai créé il y 2 ans le site officiel Wordpress du projet (www.bloodianchronicles.com) avec une partie blog où je raconte l'histoire du projet et donne des trucs et astuces et des tutos pour démarrer sous Unity.

Il faut également être présent sur les réseaux sociaux, sur les plateformes de jeux indé et créer un trailer vidéo. Il faut communiquer régulièrement pour faire connaître votre jeu, susciter l'intérêt et bâtir une petite communauté.

Communiquer vous permet également de faire de belles rencontres. C'est ainsi que Stéphane Picq, grand musicien de jeux vidéo (CRYO) a eu la gentillesse de composer le thème musical du jeu. **Figure 10**

Figure 8

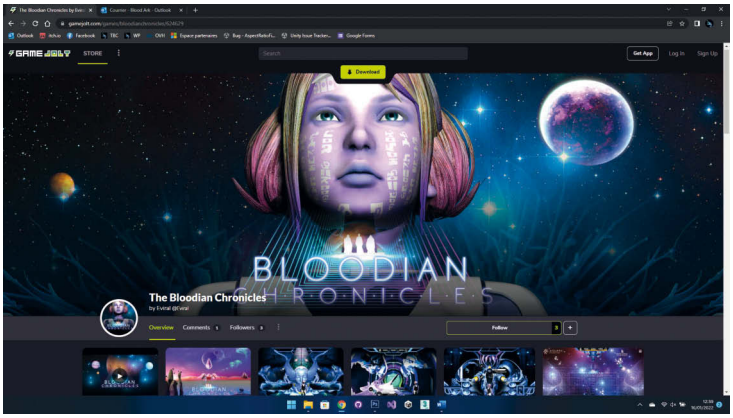


Figure 10

Figure 9

Conclusion

Développer son propre jeu est une épopée passionnante. C'est un long chemin semé de petits échecs et de petites réussites. Quand on veut quelque chose, on peut l'obtenir, à force de volonté et de persévérance. Vos limites sont celles que vous vous fixez, pensez au proverbe "Ils l'ont fait, car ils ne savaient pas que c'était impossible". Je vous sens motivé…

Alors, lancez-vous !

Florian Rival
Fondateur de GDevelop
Ex-ingénieur Google
passionné de jeux vidéo

Retour sur le projet GDevelop

Le projet open source GDevelop franchit une nouvelle étape de son développement par le biais d'une première levée de fonds, Florian Rival nous parle de son projet.

GDevelop date de 2008. C'est un outil qui a beaucoup évolué, mais toujours très centré sur l'accessibilité à la création de jeux vidéo. Bien que l'outil soit central dans le projet, il est conçu comme un écosystème dans lequel la monétisation ou encore le game design auraient une place pleine et entière. GDevelop s'adresse en majorité à des créateurs non professionnels, qu'il faut accompagner et guider en donnant des clés pour faciliter le développement. Pour gagner en vélocité en constituant une équipe investie sur le projet, la levée de fonds m'a paru être la solution la plus évidente.

L'organisation du projet

Aujourd'hui, il y a une « core team » professionnalisée mais du fait de son caractère essentiel à la vie du projet, le noyau dur de contributeurs open source est conservé. La « core team » distribuée entre Londres, la France, le Canada et le Texas garde une organisation de travail via Discord. Cela permet le travail asynchrone, même si on tente de garder la majorité des personnes qui travaillent sur la technique sur un fuseau horaire commun.

L'équipe initiale a été étendue à 3 personnes supplémentaires, que sont des « content creators » chargés de publier régulièrement des tutoriels YouTube, demandés depuis très longtemps et très appréciés des utilisateurs, ainsi qu'un Community Manager chargé de communiquer sur les réseaux et mettre en avant les créateurs qui produisent sous GDevelop.

Cette structure devrait permettre d'être plus réactif pour faire évoluer le produit et publier les correctifs. Ainsi nous avons fait évoluer des choses qui nous paraissaient trop sommaires comme les profils utilisateurs, la mise en avant des auteurs des extensions ou des jeux exemples, la mise en place de statistiques sur les jeux, et à l'avenir un travail sur l'interface et sur l'ergonomie.

La « core team » a un rôle facilitateur en orientant les contributeurs tout leur laissant la liberté, comme auparavant, de travailler sur des fonctionnalités qu'ils souhaitent intégrer à GDevelop et si besoin leur apporter un support.

L'exemple de l'extension « In App Purchase » qui est à l'initiative d'un contributeur proposée sur le repo GitHub est parlant, puisque nous avons rédigé la page d'aide et fait les modifications nécessaires pour le rendre disponible rapidement.

Notre système de priorisation est conservé par le biais de contributions et de reviews qui se font via le forum, le discord ou l'espace de discussion de GitHub sur lesquels sont soumises les idées de contribution ou fonctionnalités. Dernièrement les discussions se sont orientées pour avoir des équipes par type de contribution : une équipe pour modérer et améliorer les exemples, une autre pour les extensions, etc. C'est grâce à eux que nous avons une centaine d'exemples et d'extensions, ce qui est absolument incroyable.

Comme créateur original et développeur principal du moteur, je porte un œil sur tout ce qui se fait, mais l'idée est de pouvoir déléguer de plus en plus à la « core team » ou aux contributeurs expérimentés.

Le proche futur de GDevelop

À l'avenir, nous pensons pouvoir attaquer des sujets intéressants pour les créateurs de jeux. La monétisation étant un

sujet important, une première extension a vu le jour, mais en l'état cela reste une tâche ardue du fait de la nécessité de passer par du développement bas niveau. On cherche à rendre plus accessible la monétisation par achats intégrés sans passer par des méthodes complexes de mise en œuvre et de configuration.

Autre sujet important, encore peu avancé : la création de jeux multijoueurs qui passe actuellement par des API bas niveau utilisant web Socket ou le P2P. L'idée est d'offrir une gestion plus haut niveau du multijoueur avec un service existant, ou encore mieux, le nôtre. On parle ici des mécaniques de synchronisation des joueurs, des outils de « matchmaking », de salles ou même de tchat entre joueurs. Dans un premier temps, nous pensons offrir des tableaux de scores, ce qui permet d'ajouter un côté multijoueur/social simple à un jeu solo.

Autre sujet essentiel, la publication d'un jeu qui reste pour le novice un frein compte tenu des nombreux stores disponibles. En mettant nous même plus en avant les jeux créés avec GDevelop, les auteurs gagnent en visibilité.

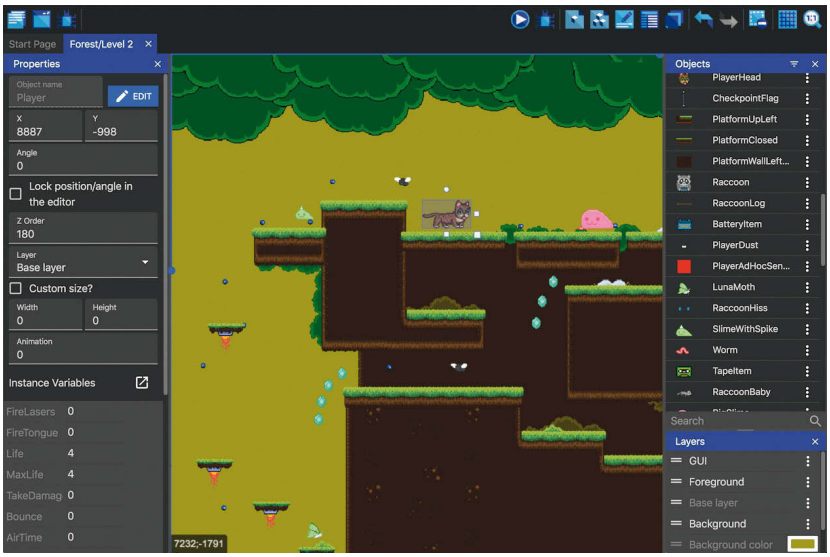
Une question récurrente des utilisateurs est celle de la prise en charge de la 3D. Pour le moment, on se concentre sur la 2D avec l'objectif d'offrir la meilleure expérience possible (et la plus accessible possible). Sur ce point la communauté peut aider pour travailler sur un support qui donnerait un peu de profondeur aux jeux avec de la 2.5D.

GDevelop sur un plan technique

GDevelop est une des premières applications à utiliser beaucoup de WebAssembly. Les classes qui définissent la structure d'un jeu, et les outils comme la transpilation en JavaScript des évènements sont écrits en C++. Compilées avec Emscripten en WebAssembly, les librairies sont ensuite utilisées par l'éditeur développé en JavaScript et React pour l'interface.

On a les avantages d'un langage robuste associés aux avantages des technologies web (cross-platform, rapidité de développement, architecture en composants). Cela nous permet de proposer une version desktop et une version web séparée qui rend GDevelop accessible sur toutes les plateformes disposant d'un navigateur moderne.

GDevelop et les extensions intégrées, codées entièrement en TypeScript, se basent sur des bibliothèques open source reconnues, notamment PixiJS pour le rendu graphique via WebGL. Il est suffisamment performant pour produire des jeux avec un rendu digne des productions d'aujourd'hui et suffisamment léger pour produire des jeux ciblant des plateformes comme les Instant Games sur Facebook ou Snapchat. On pense réécrire en WebAssembly les parties les plus critiques en termes de performance. Par exemple, gérer le positionnement dans l'espace et les collisions, afin d'avoir des performances stables et prévisibles (là où les moteurs JavaScript peuvent parfois désoptimiser du code en cas d'utilisation d'une construction particulière ou d'une erreur de type).



GDEVELOP VU PAR LES CONTRIBUTEURS

Arthur et Davy sont deux contributeurs réguliers de GDevelop et nous parlent de leur engagement sur le projet.

Arthur : quelles raisons t'ont poussé à contribuer à GDevelop ?

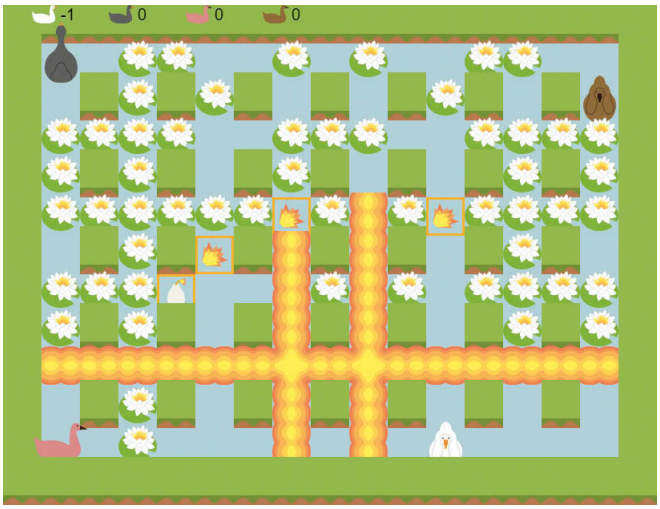
J'ai depuis le tout début été fasciné par la volonté de rendre le développement de jeux plus simple. Mon tout premier projet de programmation était un logiciel en python pour faire un jeu similaire aux "livres dont vous êtes le héros" avec de simples fichiers csv. La découverte GDevelop a été une grande révélation, fasciné par le produit, j'ai commencé à l'utiliser. Quand j'ai pris conscience du caractère open source du projet, j'ai voulu y contribuer d'une manière ou d'une autre. Cela m'a permis aussi de m'impliquer sérieusement dans un projet tout en faisant mes armes en programmation.

J'ai des idées plein la tête en tête pour de futures évolutions telles que l'ajout de fonctionnalités 3D rudimentaires, ou l'intégration du multijoueur avancé comme on peut voir sur d'autres moteurs tels que la réplication, la compensation du lag, la prédiction d'inputs, avoir un protocole léger, etc.

Davy : quelles raisons t'ont poussé à contribuer à GDevelop ?

Le 2e confinement fut une période d'ennui qui m'a donné l'occasion de faire autre chose que de coder 8h par jour pour un job alimentaire. C'était l'occasion de créer des jeux vidéo sans pour autant m'investir dans quelque chose de compliqué.

Sans trop d'espoir, je me suis intéressé aux outils de création de jeux vidéo dans le monde de l'open source. J'ai eu l'agréable surprise de découvrir GDevelop et son système d'évènements qui m'a paru limpide. En créant un jeu à la « Bomberman », je me suis posé la question de simplifier la gestion du mouvement du personnage. Je me suis donc plongé dans le code source de GDevelop pour ensuite essayer de



faire cette modification. Sans que cela aboutisse, j'ai mis le doigt dans l'engrenage qui m'a poussé à travailler sur d'autres évolutions.

J'ai été impressionné par le système d'extension. Il permet aux utilisateurs de développer des extensions avec la même logique que pour les jeux.

UNE SUCCESS STORY D'UN DÉVELOPPEUR GDEVELOP

Pouvez-vous nous en dire plus sur vous ? D'où vient l'idée de faire des jeux ?

Je m'appelle Ittalo Ornilo, j'ai 28 ans et j'étudie la communication et les médias numériques à l'université fédérale de Paraíba au Brésil. Depuis toujours attiré par les jeux vidéo, beaucoup de mes souvenirs en famille et avec des amis y sont liés. À l'université, j'ai fait la connaissance de personnes qui, comme moi, souhaitaient aller au-delà du fait de jouer. Naturellement, nous nous sommes tournés vers des enseignements de game design.

Pourquoi GDevelop et pas un autre moteur de jeu (Unity, Unreal, et consorts) ?

Je débutais et GDevelop semblait être une solution rapide, facile d'accès, gratuite et ne nécessitant pas de connaissances poussées en programmation. C'était parfait pour mettre en œuvre mes idées sur le game design.

Qu'aimez-vous dans GDevelop ?

Beaucoup de choses en fait, la facilité à créer et placer des éléments, et le système d'évènements facile à appréhender. Ce que je préfère dans le moteur est le nombre incroyable de comportements pré-codés qui peuvent être associés aux objets. Simples et efficaces, et faciles à paramétrer, on produit un jeu rapidement en quelques lignes.

L'apprentissage de GDevelop ?

J'ai beaucoup travaillé en visionnant quotidiennement les nombreux tutoriels sur YouTube notamment The Gem Dev et Wishforge Games. Aussi en m'adressant à la communauté sur le forum et le serveur Discord, et surtout avec un développeur expérimenté en JavaScript qui m'en a dit beaucoup sur le moteur.

Quand avez-vous eu l'idée de faire le jeu "Vai Juliette!" ?

J'ai fait mes premières armes avec GDevelop en créant un « endless runner » pour un ami enseignant. Juliette, une candidate Big Brother Brazil 2021 a charmé tout le pays, moi le premier. J'ai donc conçu un jeu en lien avec son image que j'ai ensuite développé avec GDevelop.

Combien de temps pour passer de l'idée à la publication ? Comment le jeu a-t-il été publié et quelle a été la réaction du public ?

Le brouillon comportait la majorité de mes idées. À partir de là, il a fallu près d'un mois pour arriver jusqu'à la publication sur Play Store, App Store et Game Jolt. Son succès est lié à la concomitance de plusieurs éléments : le fait de le centrer sur une star naissante, en faire un jeu « hyper casual », le support de la « team Juliette » et ses fans qui y ont joué en ont fait de bons retours et m'ont aidé à l'améliorer. Sans ces derniers, le succès n'aurait pas été au rendez-vous.

Et la monétisation ?

Ne voulant pas être exposé à des problèmes liés au copyright du fait d'avoir réutilisé la voix de Juliette ou son image, j'ai fait le jeu comme un « fan game », car j'adore le personnage de Juliette ! Il me semblait mal venu de profiter financièrement de ce succès.

Carrière, compétence, formation, salaire

François Tonic

Le marché du jeu vidéo en France pèse plus de 5 milliards €, ce qui est actuellement un record pour le pays. La croissance annuelle est de +11 %. 2021 devrait être aussi une année record, favorisé par la crise sanitaire, le télétravail. Plusieurs facteurs expliquent ce boom du marché : les nouvelles consoles, les usages mobiles toujours croissants, les nouveaux jeux. L'AFJV indique que les joueurs français ont acheté 27 millions de jeux ! La console pèse 50 % du marché. Le jeu desktop et le jeu mobile pèse quasiment chacun 25 %.

Contrairement à ce que l'on pourrait croire, la console reste la référence du marché et des gamers. Les ventes matérielles et logicielles se portent bien. En 2020, 2,3 millions de consoles ont été vendues (merci la PS5). Dans le même temps, le marché des accessoires se porte lui aussi très bien.

Une grande diversité de jeux, mais finalement assez peu d'éditeurs

Selon les chiffres 2020 de la S.E.L.L., FIFA 21 a généré 77 millions € de vente, Animal Crossing New Horizons, 53 millions €, Call of Duty, 43 millions €. Vous l'aurez compris, les poids lourds du marché des jeux consoles restent très concentrés sur quelques studios : Ubisoft, Sony, Nintendo, Microsft, Bandai, Electronic Art, Activsion Blizzard. Ainsi, Nintendo a vendu pour 120 millions € de jeux (rien que dans le top 20) en France. Ce sont des studios capables d'adresser le monde entier et de mobiliser des centaines de développeurs et des budgets énormes : jusqu'à 250-300 millions $. Finalement, les grands studios sont toujours un peu les mêmes depuis 15 ans. Sur la partie purement PC est un peu plus diversifié : Take-Two Interactive, Sega, CD Project Red, Capcom, Electronic Art, Microsft, Depp Silver, Ubisoft, Focus, Activision, etc.

On constate que le top 5 (Nintendo, Electronic Art, Ubisoft, Activision Blizzard, Sony) domine très largement le marché en valeur et en volume. Ils peuvent profiter de licences puissantes et lucratives.

Sur la partie mobile, le marché jeux est massivement segmenté. De petits éditeurs parviennent à se faire une place sur des gammes de jeux, par exemple les jeux de salle d'attentes. Nous retrouvons les grosses productions : Coin Master, Minecraft Pocker Edition, Clash of Clans, Fortnite, Dragon Ball Z, Candy Crush Saga, etc. Cette diversité se retrouve dans les studios. Cependant, les revenus générés se retrouvent eux aussi éclatés et les achats intégrés ne sont pas une garantie de succès surtout s'ils sont trop nombreux.

Jeux mobiles : se faire une place et faire durer le succès

Le marché mobile est particulièrement concurrentiel et le budget marketing pour promouvoir le jeu et être bien référencé dans les stores est de plus en plus élevé et les règles changent régulièrement. Aujourd'hui, pour un petit éditeur, le marché mobile est difficile et compose avec les stores et les éditeurs de ces stores. Les enjeux sont nombreux : un jeu visible, acquérir les joueurs et les garder sur la durée.

Le succès peut être fulgurant et de courte durée. Rovio a connu un succès planétaire avec Angry Birds. Mais Rovio n'a pas su se renouveller. L'éditeur a décliné Angry Birds à l'infini. Si les premières éditions spéciales étaient réussies, les développeurs ont usé la licence et inutilement complexifié le gameplay. L'éditeur annonce un remake complet de son jeu pour 2022 : développement from scratch, utilisation d'Unity, refonte du gameplay, etc. Pour l'éditeur, c'est un pari risqué mais faute d'alternative, il n'avait pas le choix.

L'autre succès est Monument Valley. UsTwo a su concevoir un jeu surprenant, captivant et visuellement sublime. Contrairement à Rovio, UsTwo n'a jamais voulu user la licence en multipliant les éditions. C'est même tout l'inverse : des évolutions peu nombreuses, une mise à jour avec de nouveaux niveaux pour Monument Valley 2 sortie en octobre 2021, 4 ans après la sortie du jeu ! Le jeu est vendu plus cher que la moyenne, et avec assez peu de niveaux. Le titre se rattrape sur l'ambiance, le gameplay et le visuel.

	Budget	Durée du développement	Revenu (dans les 12 mois)
Monument Valley (2014)	852 000 $	13 mois / 8 développeurs	+5 millions $ +14 millions $ (2 ans)
Forgotten Shores (extension)	549 000 $	7 mois	?
Monument Valley 2 (2017)	2,2 millions $	18 mois / 16 développeurs	+ 10 millions $
The Lost Forest (extension, 2021)	?	?	?
Soit	3,6 millions $	-	29 millions $

Monument Valley 3 est en développement depuis 2019. UsTwo prend donc son temps pour proposer une expérience immersive. Le succès du jeu s'appuie sur le gameplay, l'ambiance, mais aussi la rareté.

Contrairement à Rovio et Angry Birds, Candy Crush a su résister à l'usure. Le jeu est sorti il y a 10 ans, mais il reste toujours dans le top des jeux mobiles les plus joués ! Et surtout, il est très rentable : 1,2 milliard $ de revenus en 2021 ! Finalement Candy Crush Saga est l'exemple réussi d'un jeu addictif.

Sur mobile, les genres de jeux sont nombreux. Les plus importants dont : les jeux casual, puzzle, simulation, arcade, action.

Des profils nombreux !

Développeur de jeux ? Le terme est aussi vague qu'imprécis. Le jeu vidéo exige de nombreuses compétences quelle que soit la taille du studio : développeur, animateur 3D, graphiste, game designer, directeur artistique, tech lead, designeur UX, designeur des personnages, responsable de la monétisation, etc. Les profils les plus recherchés dans les annonces de janvier 2022 étaient : développeur C#, développeur gameplay, testeur, designeur game, développeur mobile.

Les offres sont très diversifiées et dans la France entière. Les studios se situent à Paris, Lyon, Bordeaux, Aix-en-Provence, Montpellier, etc.

Un jeu se développe comme tout projet IT : une équipe à multiple compétence, un tech lead, un chef de projet. Le jeu combine de nombreux assets (graphisme, 3D, gameplay, son, etc.). Pour le développeur de jeu, les compétences techniques sont essentielles. La passion ne suffit pas.

QUI DÉVELOPPE QUOI ET À QUEL PRIX ?

AAA : c'est la grosse production. Plusieurs années de production, des centaines de personnes, un budget pouvant dépasser les 100 millions $. Les AAA sont produits par les grands studios et les constructeurs – éditeurs de consoles.

Petits studios / studios moyens : une dizaine, quelques dizaines de développeurs. Les petits studios sont les plus nombreux. Ils produisent des jeux pour desktop, mobiles, etc. Ils peuvent investir de nouveaux marchés.

Indépendant : souvent des développeurs seuls développant pour le plaisir ou pour produire quelques jeux. Ils peuvent se faire aider par des développeurs, spécialistes externes pour tel ou tel asset.

Quel est le budget pour développer un jeu ? Ne parlons pas des jeux AAA. Pour un dév indépendant, le budget doit être strictement limité. Il ne doit pas dépasser 2 000 – 3 000 € pour le matériel, les licences, les assets, quelques interventions internes sur des points précis de code ou de graphisme par exemple. Le dév indépendant travaille sur son temps libre sauf s'il en fait son job, mais dans ce cas, il faut être capable de créer régulièrement de nouveaux jeux. L'indépendant peut se rembourser en mettant en place un modèle économique à son jeu.

Pour le petit studio, la situation varie. Un studio de -20 développeurs peut produire des jeux à 800 – 900 000 $ sur 18 à 24 mois, tout dépend des ambitions et des capacités à supporter de tels budgets.

Les studios spécialisés en jeux mobiles de type « salle d'attente » doivent produire des jeux à la chaîne. Ce sont des jeux qui ont souvent une durée de vie limitée. Ils doivent être faciles à jouer et à comprendre. Les parties doivent être courtes. Ce sont aussi des jeux générant des revenus à grande échelle (x millions de téléchargements), car souvent ils n'ont pas d'achats intégrés et sont gratuits. Le budget moyen d'un casual game ne doit pas dépasser 30 – 40 000 €, idéalement 25 – 30 000 €. L'équipe de création doit pouvoir alimenter les développeurs en idée tous les mois.

Fin 2020, Kevurugames publiait les fourchettes suivantes :

- Casual game 2D simple avec peu de niveaux : - 10 000 $
- Casual game 2D : 25 à 50 000 $. La 3D est possible, mais peut engendrer un surcoût
- Jeux 3D avec graphismes avancées : jusqu'à 120 000 $
- Jeux multijoueurs 3D : minimum 150 000 $

Si vous voulez faire de l'animation 3D, il faut maîtriser un ou plusieurs moteurs tels que Unreal, Unity. Ces moteurs sont complexes et exigent une maîtrise des outils et des langages. On développe en C++ ou en C#. Par contre attention, les exigences techniques peuvent être très très longues.

L'industrie du jeu vidéo en France

Source : étude sur l'industrie du jeu vidéo en France. Étude publiée en mars 2021, mais situation de 2018. Nous n'avons pas trouvé d'éléments plus récents.

En France, le marché est très éclaté :

- 960 entreprises dont 575 studios
- Presque 12 000 emplois directs
- + 40 % des entreprises / studios sont à Paris et sa région, mais plusieurs régions françaises sont très actives (Occitanie, Nouvelle-Aquitaine, Auvergne Rhône Alpes).
- 21 % de femmes dans les formations dédiées (chiffre de 2021)
- +50 formations au jeu vidéo dont 2 formations publiques (CNAM et université Côte d'Azure)

La France est un des pays les plus actifs. C'est le 2e pays producteur de jeux en Europe, devant l'Allemagne, mais loin derrière l'Angleterre, poids lourd historique du jeu. Le Canada héberge moins de studios, mais les équipes sont énormes et totalisent + 28 000 emplois !

Côté salaire, il est très difficile de se faire une idée précise. Beaucoup de jeunes développeurs rêvent de travailler dans le jeu ce qui semble faire baisser le salaire moyen en France. Nous avons constaté qu'un développeur débutant peut espérer un salaire moyen de 2 000 € (la fourchette est cependant assez large). Un développeur senior sera vers 3 500 – 4 500 € selon le profil, les compétences. Quand on regarde les chiffres, on constate une disparité très grande : allant de 20 – 22 000 € à 35 – 40 000 € pour un développeur débutant. Un profil senior peut aller de 50 à 90 000 €. La taille du studio, le type de jeux jouent sur les salaires.

Les jeux mobiles de type salles d'attente sont des jeux qui doivent être rapidement développés à des budgets serrés (20 – 30 000 €) et en quelques semaines, ce qui influencent directement les salaires proposés. Ce sont des jeux avec des durées de vie courtes. Et les studios indépendants sont parfois obligés de multiplier les jeux pour multiplier les revenus et espérer qu'un ou plusieurs titres rencontrent un succès, surtout si le jeu est gratuit, sans achat intégré. La publicité sera alors le seul revenu direct, nécessitant de nombreux joueurs pour rentabiliser le développement.

QUELQUES ÉCOLES DE JEUX

- **CNAM Enjimin** à Angoulême
- **Gobelins à Paris :** cette école reste une référence
- **Institut de l'Internet du multimédia** à Paris
- **Supinfogame-Rubika** à Valenciennes
- **Isart Digital** à Paris
- **Créajeux** à Nîmes

Liste non exhaustive. Un conseil : formez-vous, formez-vous, formez-vous ! N'hésitez pas à créer des PoC, à participer à des projets. Régulièrement, les studios veulent voir vos réalisations.

La guerre des plateformes de streaming de jeux

François Tonic

Depuis 2 ans, c'est une guerre qui s'étend et s'amplifie : le stream gaming. Pour faire simple, ce sont des jeux streamés depuis une plateforme d'accès unique. Ce streaming est accessible sur mobile, desktop, tablette, et même la télévision ! La guerre est d'importance, car Sony, Google, Microsoft s'y mettent. Sans oublier Apple et même Netflix et bientôt Amazon !

Ce nouveau modèle s'appuie sur la notion de cloud gaming, et de GaaS. Ces plateformes s'appuient sur les services cloud et donc de milliers de serveurs dans les datacenters. Pour pouvoir streamer les jeux à des millions de joueurs, il faut des ressources colossales pour provisionner les instances de chaque joueur, stocker les parties, réduire la latence et donner la meilleure expérience de gaming. Ainsi, Google avait annoncé à la présentation de son service Stadia, que la plateforme s'appuie sur des CPU et GPU adaptés, un stockage SSD, et jusqu'à 16 Go de RAM par instance. Pour pouvoir jouer à des jeux premium (de type AAA donc), il faut beaucoup de puissance.

L'enjeu est d'attirer les jeux les plus importants du marché et de convaincre les développeurs et les studios de porter les jeux sur les plateformes, ce qui rajoute un développement supplémentaire.

Le marché est tellement prometteur que Netflix a lancé une offre, pour le moment très limitée et Amazon a présenté son service : Amazon Luna (projet Tempo). Tout naturellement, Luna repose sur les infrastructures et services AWS. Le service supportera Windows, macOS, Fire TV, iOS, Android. Une manette Luna a été créée pour le service.

Comment développer pour du stream gaming ?

Déployer son jeu sur une plateforme de cloud gaming / GAAS / stream gaming n'est pas aussi simple que de cliquer sur un bouton. Il faut adapter a minima, supporter l'architecture et les instances cloud, gérer l'authentification imposée, etc. Tout développeur de jeux ne sera pas autorisé à support ces services.

Prenons l'exemple de Google Stadia. Vous devez d'abord être développeur Stadia référencé et officiel. Il faut pouvoir supporter les instances imposées par Google, supporter et implémenter la pile logicielle (Debian, API Vulkan, SDK dédié Stadia). Stadia supporte Unreal Engine, Unity et de nombreux outils tels que Havok, RenderDoc, Visual Studio, LLVM, FaceFX, etc. Cela signifie que tous les jeux utilisant des langages et moteurs non supportés ne pourront pas être déployés sur la plateforme. D'autre part, votre demande de référencement doit être approuvée par Google. Le service n'est pas gratuit : vous reversez à l'éditeur un % sur les revenus générés. Pour inciter les développeurs et studios à venir, Google a décidé de réduire sa commission.

Si on regarde côté Microsoft et Xbox Cloud gaming (xCloud), le service permet de publier les jeux sur Windows, Xbox One, Xbox Live pour iOS et Android. Là encore, il faut optimiser le jeu, notamment dans un usage mobile.

Bref, vous l'aurez compris : tout le monde veut son service de stream gaming. Plus que jamais, les services savent que le succès passera par le contenu, donc les jeux. Il faut convaincre les développeurs et les studios. Certains services misent avant tout sur les grosses productions, mais des services comme ceux d'Amazon, Microsoft, Apple, Google doivent convaincre les studios indépendants. Le potentiel est là, reste à le concrétiser.

LES PRINCIPALES OFFRES

Google Stadia

Malgré un début très difficile, Google persiste. Ainsi, en janvier dernier, c'est le superbe Darksiders III qui avait été annoncé. L'éditeur a fermé le studio dédié à Stadia, faute de succès. 2022 sera sans doute déterminant pour l'avenir du service.

Microsoft Xbox Cloud Gaming

Avec la Xbox, Microsoft possède une des consoles les plus populaires du marché. L'éditeur investit sur le cloud gaming avec le projet xCloud (Xbox Cloud Gaming).

Apple Arcade

Le lancement raté d'Arcade est toujours dans les esprits, mais peu à peu, Arcade a su étendre son offre avec des jeux haut de gamme et de nombreux jeux indépendants. Arcade veut adresser tous les joueurs. L'idée est d'étendre l'expérience iOS avec les jeux. Apple veut des portages et des exclusivités. Aujourd'hui, plus de 200 jeux sont disponibles. Arcade a du mal à exister face aux autres services et pourtant, il ne manque pas d'atouts.

Netflix Jeux / Gaming

Disponible en France depuis novembre 2021. Offre très restreinte à son lancement : 10 jeux (décembre 2021). Accessible directement depuis l'application Netflix (Android / iOS). Service sans coût supplémentaire pour l'abonné.

Nvidia GeForce NOW

Le constructeur de GPU propose aussi son service de stream, GeForce NOW pour desktop et Android. Le constructeur propose des SDK et outils dédiés pour adapter les jeux et les optimiser. Le service est compatible avec Steam.

PlayStation Now

Sony surfe sur le succès de sa console et surtout sur son catalogue parmi les plus excitants du marché ! Mais le service est disponible uniquement PS4, PS5 et Windows.

Franck Dubois
Video Game Codeur
Développeur agile
Formateur JavaScript / Unity / GDevelop

Les ressources gratuites ou presque pour ceux qui bossent seuls.

L'écosystème de la création de jeux englobe de nombreux métiers techniques et artistiques. Du côté technique, la partie la plus visible est le développement. Pour ce qui est de l'artistique, nous trouvons les bruitages, la musique, la typographie, les graphismes.

Il est rare de posséder tous les profils. Lorsque l'on est développeur, et uniquement développeur, plusieurs solutions sont possibles pour les autres éléments. La première est de faire soi-même, pas simple si comme moi, vous n'êtes pas très bon. La deuxième est de faire appel à un professionnel qui fera ce que vous attendez, avec un coût. Troisième option : trouver des ressources sur le net gratuites ou payantes. Je vous propose plusieurs ressources d'assets.

Humble Bundle (https://fr.humblebundle.com)
HB vend toutes sortes de choses, plus ou moins éloignées du jeu vidéo avec la possibilité de supporter une œuvre caritative. Nous y trouvons des ressources graphiques ou sonores sous licence d'excellente qualité à des prix très intéressants.

Itch.io (https://itch.io/)
Un peu fourre-tout puisqu'on y trouve des jeux ainsi que des assets classés par thématiques payants ou gratuits avec un moteur de recherche pour vous faciliter le travail. La qualité des contenus gratuits est variable.

Craftpix
Les assets sont payants ou gratuits. Les tarifs sont raisonnables : un peu moins de 36€ par an. Vous avez un accès illimité à toutes les ressources disponibles. Un rapport qualité/prix imbattable. **Figure 1**

Kenney.nl (référencé aussi chez itch.io)
Plus limité en termes de contenu gratuits ou payants sous forme de bundles.

Figure 1

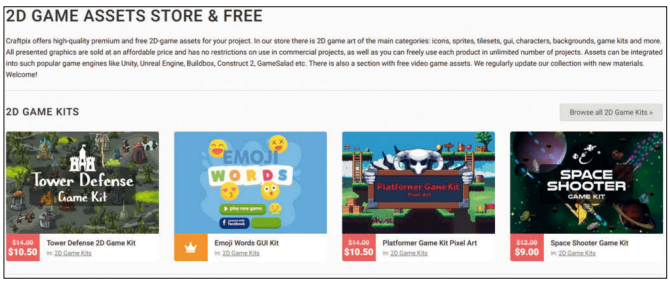
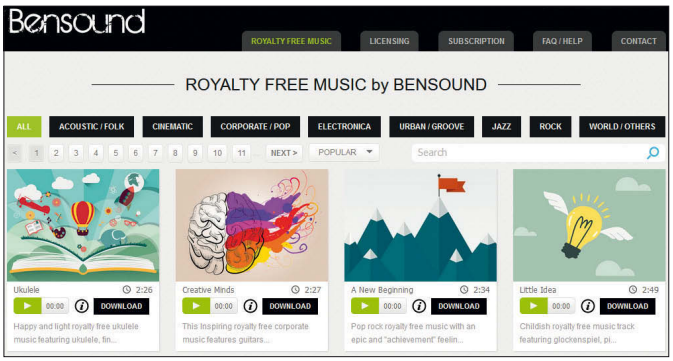


Figure 2



The Spriters Resource (https://www.spriters-resource.com)
Si vous êtes fan de jeux et assets old school, vous y trouverez votre bonheur puisque vous avez accès à des contenus extraits d'anciens jeux classés par plateforme et genre. Gratuit pour les projets personnels et non commerciaux.

Universal LPC Spritesheet Generator (https://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator/)
Un générateur de personnage animé dont vous allez pouvoir sélectionner les attributs de la couleur des yeux à celle des chaussures, humain ou pas. Bref, il y a de quoi faire.

Pour les textes, boutons et polices de caractères

La création de polices de caractères est un travail titanesque qui nécessite des compétences particulières. Mais grâce à la magie du net, il y a de quoi trouver son bonheur sans trop d'efforts. Da button factory (https://dabuttonfactory.com/fr/) est un générateur de boutons un peu sommaire. Il fait le job si vous n'avez pas trop d'exigences. Dans la même lignée, Cooltext (https://cooltext.com/) en met plein les yeux avec beaucoup de formats disponibles. Dafont (https://www.dafont.com/fr/) une bibliothèque de polices qui devrait ravir les créateurs les plus exigeants. Faites tout de même attention à la licence, si vous en faites un usage commercial.

Pour les sons et musiques

Soundbible (https://soundbible.com) et Universal Sound Bank (http://www.universal-soundbank.com/) proposent respectivement une banque de sons parfois étonnants et de nombreux sons comprenant bruitages, musiques, samples, ambiances de qualité variable. Vous devriez y trouver ce qu'il vous faut. Pour la musique, jetez un œil à Bensound (https://www.bensound.com/) avec ses nombreuses musiques gratuites ou payantes et classées par thématique. Autre site bien connu : freesound.org. **Figure 2**

Les outils

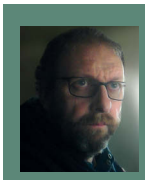
Plutôt que de digérer la documentation de Gimp ou de Photoshop pour faire des petites modifications, il existe des outils aux réglages simples, faciles à maîtriser.

XnView (https://www.xnview.com/fr/xnviewmp/) permet de faire du traitement d'images en lot comme redimensionner, recadrer, appliquer des filtres ou bien créer/découper des planches de sprites. La licence est gratuite pour un usage personnel. Pour les nostalgiques du pixel art, Piskel (https://www.piskelapp.com/) est un outil en ligne pour créer des sprites animés.

Enfin Audacity, éditeur de sons multipistes, pour retravailler vos effets sonores.

Les modèles économiques du jeu vidéo

Si vous êtes indépendant, vos revenus sont directement liés à vos productions et aux revenus qu'elles engendrent. Différentes solutions ou modèles économiques sont possibles pour les générer, certains réservés à ceux qui ont de gros moyens, d'autres accessibles aux plus modestes. L'argent est le nerf de la guerre.



Franck Dubois
Video Game Codeur
Développeur agile
Formateur JavaScript / Unity / GDevelop

La vente

Modèle historique datant des années 70 : vous achetez votre jeu tout comme vous acheteriez un DVD. Le modèle a évolué avec l'arrivée du net où le dématérialisé est devenu une norme : vous achetez votre jeu, vous le téléchargez et votre achat est stocké sur les serveurs du revendeur. Ce marché reste difficile notamment pour les consoles.

La publicité

Proche de ce qui se fait sur le web dont le principe est de tirer des revenus en mettant à disposition des annonceurs des espaces dédiés. En continue et en surimpression sur l'écran de jeu sous forme de bannière, ou en intersticiel pleine page, entre introduction ou niveaux, le choix vous appartient. Du fait de la faible rémunération du modèle, tirer des revenus conséquents exige que votre jeu vidéo bénéficie d'une forte base de joueurs. Cela nécessite de passer par une régie, une des plus connue étant AdMob, qui vous propose une panoplie d'outils pour suivre vos revenus. C'est un modèle techniquement facile à intégrer.

L'abonnement à une licence

Le jeu World of Warcraft est la référence dans le domaine avec des chiffres qui parlent d'eux-mêmes et environ 8 millions d'abonnés en 2020. Ce modèle vise à fidéliser les joueurs par le biais de mondes persistants et de contenus additionnels toujours plus nombreux. C'est un modèle à gros moyens puisque le parc de serveurs doit être suffisamment dimensionné pour une expérience de jeu optimale. Dans la même veine, ankama a créé Dofus, son MMORPG par abonnements.

Le free to play et les achats intégrés (item-selling)

Le free to play est un modèle de jeu gratuit aux fonctionnalités limitées mais pas trop pour une bonne adhésion où l'accès aux niveaux supérieurs peut être possible mais difficile. L'argent rentre ici par la vente d'accessoires pour une somme modique qui vont de la simple esthétique comme habiller son héros à l'ajout d'attributs rendant la progression du joueur plus facile. League of Legends en est un exemple avec 1,6 milliard de $ générés en 2016 pour une communauté composée de 100 millions de joueurs. L'intégration de ce modèle ne représente pas de difficultés particulières.

Le free to play et le freemium ou pay-to-win

On reprend le modèle du jeu gratuit où la progression, les contenus ou les niveaux bonus peuvent être obtenus après de très longues heures de jeu et dont l'accès est rendu immédiat.

Les extensions payantes ou DLC (Downloadable Content)

Parfois gratuites, souvent payantes, ou en « season pass », les DLC rallongent la durée de vie d'un jeu vidéo en proposant des packs d'extension incluant de nouvelles cartes, de nouvelles armes ou encore des skins supplémentaires. Les prix sont variables parfois proches de celui du jeu. La plupart des éditeurs jeux en proposent parmi lesquels Dead Cells, ou Borderlands. Du fait de leur qualité, souvent annoncés et attendus par les joueurs, ils sont une autre manière de les fidéliser.

Les abonnements à un service (Stadia, PC/XBox Game Pass)

Tout comme Netflix ou Prime Vidéo propose films, série et docs à la demande à un tarif très abordable, les éditeurs ont adapté ce type de service au jeu vidéo : des jeux en illimité pour une somme modique. Contre un abonnement, vous avez accès sans limite à un catalogue : vous téléchargez, installez et jouez. Les contenus s'enrichissent à coups de rachats des éditeurs aux grosses licences porteuses de valeur. Le dernier étant celui d'Activision par Microsoft pour 68 milliards de $, dont ce n'est pas le 1er fait d'arme. On assiste donc à un secteur qui bouge avec une tendance à la concentration. En espérant que cela n'ait aucun impact sur la qualité des futures productions ainsi acquises, à renouveler les genres.

Cloud Gaming

Un service proche du modèle précédent à la différence qu'ici vous jouez en streaming. La puissance nécessaire pour faire tourner un jeu est déportée sur des serveurs distants. L'utilisateur n'a plus besoin d'une machine surpuissante pour jouer aux jeux les plus gourmands en puissance, avec en contrepartie la nécessité d'avoir une connexion internet suffisamment rapide et munie d'une latence faible : la fibre est quasiment incontournable pour un gameplay optimal. Microsoft investit ce marché et en propose une version beta dans son Xbox Game Pass Ultimate pour jouer aux jeux Xbox depuis PC, mobiles ou tablettes.

Hyper Casual Game

Ici, ce n'est pas un modèle économique pur et dur mais surfe plutôt sur la vague de joueurs mobiles qui ont émergé depuis la crise covid. Vu leurs nombres et les prévisions, cette cible devrait générer les revenus les plus importants dans les 10 prochaines années. Un simple coup d'œil aux classements des jeux téléchargés sur le Play Store ou l'Apple Store suffit à voir la tendance. Ce public zappeur est constitué de joueurs occasionnels qui n'investissent pas de temps. Ils ne veulent ni payer (du moins dans un premier temps), ni se poser de questions avec des commandes intuitives dès que la partie débute. Nul besoin ici de miser sur un gameplay complexe, le jeu étant occasionnel, les parties doivent être courtes. Ce modèle est parfait pour entrer dans des cycles de développement courts : une opportunité pour le créateur qui ne s'engage pas sur des projets à long terme souvent synonymes de lassitude dans les petites équipes. Vous produisez en nombre et en tirer des revenus issus de la publicité, première marche de la monétisation.

Franck Dubois
Video Game Codeur
Développeur agile
Formateur JavaScript / Unity / GDevelop

Les moteurs et API les plus actifs et ceux à surveiller

Une des questions cruciales qui se pose à tout créateur de jeux vidéo est celle du moteur. Ils sont nombreux et le choix n'est pas toujours facile. Pour celui qui n'a jamais vu une ligne de code et à l'opposé celui qui maîtrise un ou plusieurs langages, le choix devrait être trivial. Eh bien non ! Le pragmatisme lié aux défis techniques, aux objectifs que se fixe le développeur, telles que la typologie du jeu ou les plateformes supportées, doivent guider le choix.

Quelques questions à se poser quand on choisit des technologies et qui influencent donc le jeu à coder :

- des assets et du code : quelles cibles ? (consoles, desktops, web, mobiles, etc.)
- Performances et qualité des rendus
- Outils et plug-ins
- Communauté, évolutions
- Coût des licences

Unity

Unity est probablement le moteur dont l'aura est l'une des plus grandes puisqu'il peut quasiment tout faire : la 3D mais aussi la 2D, du jeu PC (Windows, Mac, Linux) ou mobile (Android, iOS) et même de la VR. Une énorme popularité et des tutoriels et chaînes YouTube dédiées par milliers ! Son asset store offre toute une panoplie de scripts prêts à l'emploi, de composants graphiques ou d'outils associés sous forme de plug-ins gratuits ou payants qui vous éviteront de réinventer la roue. Bref, il y en a pour tous les usages. Ceci dit, l'éditeur reste une machine lourde digne du centre de contrôle spatial de Kourou et nécessite à lui seul une certaine patience pour le maîtriser. N'oublions pas le C#, langage utilisé qu'il faut maitriser dans une certaine mesure pour pouvoir exploiter toute la puissance de l'outil. La licence est gratuite jusqu'à 100 000 € de revenus générés sur les 12 derniers mois.

Unreal Engine

Conçu en 1998 et édité par Epic Games, Unreal Engine doit son nom au jeu Unreal : il est donc un moteur qui cible avant tout la 3D. Avec son système Paper2D, la 2D reste possible. D'une notoriété au moins égale à celle de Unity, et reconnu pour son excellent rendu graphique, il intègre tout ce qui se fait de mieux dans le domaine notamment avec Persona son système fin d'animation de personnages. A l'instar de Unity, UE a son lot de plugins prêts à l'emploi. Il n'en reste pas moins un outil dont la maîtrise technique nécessite un investissement non négligeable à laquelle s'ajoute la maîtrise du C++. L'usage du C++ permet à UE d'afficher des performances souvent au-dessus d'Unity.
UE n'est pas en reste pour ce qui est de la licence avec un reversement de 5% des revenus au-delà d'1 million $ par jeu.

Gobot

Godot est un environnement open source sans restriction que votre projet soit personnel ou professionnel. Godot dispose d'un asset library mais moins fournie et d'une communauté plus jeune et plus réduite. Cela peut être un inconvénient quand on bloque sur un bug ou un problème technique. Même si cela nous oblige à fouiller par soi-même. Pour ce qui est des langages, Godot supporte GDScript proche du python, le C#, C++ et Visual Scripting.

O3DE

Open 3D Engine est un nouveau venu dans la course aux outils de création pour les jeux AAA et multi-plate-forme. Open source sous licence Apache 2.0, O3DE est supporté par Amazon, Adobe ou encore Intel. Il offre la possibilité de coder en LUA ou d'utiliser son système de visual scripting Script Canvas. LUA n'est pas une nouveauté dans le domaine puisque Love2D framework dédié à la création de jeux vidéo en 2D en fait son langage de prédilection.

LE VISUAL SCRIPTING, C'EST QUOI ?

Connu sous le nom de Bolt pour Unity ou Blueprint pour UE, ils ont été conçus pour lever les barrières des langages. Par essence visuel, la lecture et la compréhension en sont aisées pour celui qui en voit un pour la première fois. Si on omet le sens pur de l'algorithmique, la programmation devient enfin accessible aux débutants, que vous soyez graphiste, game designer ou quoi que ce soit d'autre. Mais (hé oui il y en a un), tout n'est pas rose. Comparé à du vrai code, son usage est souvent plus chronophage que du code traditionnel, et plus complexe à modifier. Bref, ça peut vite devenir un gros bordel s'il n'est pas maîtrisé. Les puristes leur préféreront le code pur et dur. Donc ça fonctionne si on se limite à créer des jeux simples, au-delà vous allez perdre du temps et probablement vite abandonner voire détester.

MONOGAME

XNA est une architecture Microsoft visant à faciliter la création de jeux pour ses plateformes PC, consoles et mobiles adressable uniquement en C#. Abandonné en 2014, XNA a repris vie par le biais du projet open source MonoGame en y ajoutant un support cross-platform plus étendu intégrant notamment Sony avec la Playstation, Nintendo avec la Switch, Android ou Ios. Infinite Flight ou encore Streets of Rage 4 ont été développés sur ce moteur. Que ce soit en 3D ou en 2D, MonoGame rivalise sans sourciller avec Unity, UE ou Godot sans toutefois avoir leurs puissants éditeurs. Une communauté présente, mais avec des ressources réduites. Vous mettrez les mains en C#, ce qui peut être rebutant pour le débutant. L'usage en est gratuit.

Bien que la 3D soit plébiscitée dans le monde de la création du jeu vidéo, il faut garder en tête que les jeux AAA sont souvent développés par des équipes riches en profils experts aux tailles sans communes mesures avec celles d'indépendants. Pour du jeu AAA, penser pouvoir jouer dans la même cour n'est que pure illusion : la 3D peut aussi s'envisager sur des projets plus légers. A priori plus simple à coder, un jeu 2D n'est plus un choix par défaut, il est assumé et choisi pour des raisons liées au gameplay ou purement artistiques. Et il existe de nombreux outils dédiés à ce format.

GDevelop

Basé sur PixiJs, GDevelop est gratuit et open-source avec son propre éditeur traduit dans de nombreuses langues. On s'affranchit du langage sans se soustraire à une logique de code par le biais d'un système de comportements et d'événements sous forme condition / action. Il offre la possibilité de produire des contenus cross-platform avec des livrables générés automatiquement. Beaucoup de documentation, de tutoriels, mais quasi inexistants en français. Le multijoueur est possible, mais inaccessible à celui qui n'est pas familiarisé avec JavaScript. Avec des mises à jour nombreuses (trop ?), l'outil s'enrichit à chaque version grâce à une communauté très active comptant de nombreux participants. L'avenir est assuré avec des investisseurs arrivés en juin 2021.

Construct

CONSTRUCT

Multi-plate-forme, Construct 3 édité par Scirra fonctionne à peu près sur le même modèle que GDevelop avec un système de conditions / actions et des comportements pré-codés et l'usage du JavaScript. Les fonctionnalités sont nombreuses avec notamment l'intégration de la physique, de shaders pour le rendu et la gestion du multijoueur simple à implémenter. Bien qu'une version gratuite très limitée existe, pour avoir le droit d'exploiter l'outil, il faut payer la licence.

GameMaker

GameMaker Studio 2™

Avec Game Maker Studio 2, l'ambition est de rendre la création de jeu cross-platform rapide, et facile avec un éditeur complet. Possiblement multi-joueurs, pour coder avec GMS2, le langage GML ou son éditeur de scripts visuels DND sont proposés. Opera y croit fortement puisque qu'il a racheté l'éditeur yoyogames pour 10 millions de $ et en a profité pour y associer Opera GX avec sa plateforme de partage GXC. Pour un déploiement via GXC, c'est gratuit. Au-delà, si vous souhaitez produire du jeu pour PC, web mobile ou console, il faut payer une licence.

LES MOTEURS POUR LES JEUX WEB

PixiJs et Three.js

PixiJS three.js

PixiJs et Three.js ne sont pas à proprement parler des moteurs de jeu. Ce sont des API graphiques visant à simplifier l'usage de WebGL. Elles n'offrent pas nativement ce que l'on attend d'un moteur : détection des collisions et physique. Il va falloir les implémenter vous-même ou faire appel à des librairies tierces telles que ammo.js ou cannon.js. A noter que PixiJs embarque des interactions minimalistes avec la souris ou le tactile.

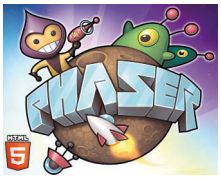
Babylon.js

Moteur 3D qui va un peu plus loin que les 2 précédents puisque qu'il intègre les mécanisme de collision et intègre une interface sur laquelle vous pouvez greffer le moteur physique de votre choix.

PlayCanvas

Avec son éditeur intégré au navigateur, le moteur PlayCanvas est complet pour créer des jeux 2D/3D cross-platform. Avec la licence gratuite, vous créez vos jeux et ils sont hébergés par l'éditeur. Tout en gardant la propriété de vos œuvres, vous en autorisez l'usage non commercial par PlayCanvas. Cette autorisation disparait avec la souscription à l'offre payante.

Phaser.js



Avec Phaser.js, on voyage léger : pas d'éditeur juste un moteur exploitable en javascript. Physique et autres classiques du genre y sont nativement intégrés. Visual Studio Code suffit pour programmer un jeu vidéo rendant la prise en main rapide. Gratuit et open source, il évolue régulièrement.

Franck Dubois
Video Game Codeur
Développeur agile
Formateur JavaScript / Unity / GDevelop

L'intelligence artificielle (IA) dans le jeu vidéo

L'IA est une discipline très à la mode de nos jours, mais pas récente : il y a plus de 60 ans, on en parlait déjà. Les lecteurs d'Isaac Asimov seront déçus, car la question a souvent été réduite au fait qu'une machine puisse penser. De ce point de vue, l'usage du terme IA est galvaudé. En effet, l'IA n'existe pas : une machine n'est ni consciente de sa propre existence, ni même capable de penser comme un humain.

Alors qu'est-ce donc l'IA ? Tout simplement des algorithmes. Big Data et Machine Learning, 2 domaines de l'IA intimement liés, en sont très friands. Pour qu'une machine puisse prendre des « décisions », il lui faut apprendre (Machine Learning) et donc amasser un nombre incalculable de données (Big Data). Traiter un nombre de données gigantesques nécessite une puissance de calcul gigantesque.

Et dans le jeu vidéo ? Une machine de jeu n'a ni cette puissance ni cette capacité de stockage. L'IA doit donc composer en matière de puissance de calcul, de mémoire et de temps d'exécution. Les algorithmes qui tentent de simuler des comportements supposés intelligents, mais pas uniquement sont donc malins.

Les techniques d'IA utilisées dans la majeure partie des jeux restent rudimentaires. La plupart des jeux étant à progression linéaire, la limite des interactions possibles permet de restreindre l'intelligence nécessaire au PNJ, sans pour autant nuire à sa crédibilité. Usant d'un socle de mécanismes de représentation des connaissances qui ont une vingtaine d'années voire plus. L'IA n'étant qu'une illusion, un choix s'impose : éviter toute méthode difficile à maîtriser, parce que complexe à mettre en œuvre.

Cela commence par des comportements très simples qui ne tiennent pas compte de leur environnement : que le joueur soit présent ou pas, le personnage non joueur (PNJ) fera toujours la même chose. On trouve ce type d'approche très tôt dans les jeux de plateforme avec des PNJ qui se déplacent sans discernement de haut en bas, de gauche à droite et vice-versa.

Une approche plus évoluée, mais pas nécessairement plus complexe à mettre en œuvre, va rendre le PNJ sensible à la présence du PJ.

Aujourd'hui des chercheurs s'amusent à travailler ce sujet avec entre autres Giorghios Yanakis et Julian Togelus qui classent la relation de l'IA avec le jeu vidéo sur la base de 2 critères. Le 1er étant centré sur le rôle de l'IA dans le jeu en tant que joueur à part entière ou en tant que contrôleur de l'environnement. Le 2d étant centré sur l'objectif que poursuit l'IA : cherche-t-elle à gagner la partie ? Ou cherche -t-elle à améliorer l'expérience de jeu ?

Une IA endossant un rôle de joueur pour gagner, qu'on retrouve dans des jeux tels que les échecs ou le jeu de Go, part du postulat que l'adversaire jouera le meilleur coup possible. À partir d'un état du plateau, elle construit un arbre des coups possibles duquel sortira celui qui contrera le meilleur de l'adversaire. L'usage de l'algorithme MinMax est courant dans ce type de jeu. La mémoire et la rapidité de l'IA étant ici déterminantes.

Ce fut une IA très médiatisée avec notamment Deep Blue vainqueur en 1997 du champion du monde d'échecs Garry Kasparov, ou encore Lee Sadol battu en 2016 par Alphago une IA dédiée au jeu de Go. Il a depuis pris sa retraite considérant futile la lutte contre l'IA qu'il considère perdue d'avance.

Pour un éditeur, cette catégorie n'est pas la forme la plus intéressante, car produire une IA qui gagne sans arrêt n'a pas d'avenir commercial du fait de la frustration qu'elle peut générer chez le joueur. Intégrée dans des boucles de gameplay avec un objectif, un défi et une récompense, rappelons que l'IA n'a pas vocation à gagner, mais plutôt de challenger les joueurs, de servir l'expérience de jeu.

Certains éditeurs ont tenté cette approche, l'une des plus frappantes fut celle de Mortal Kombat – The Ultimate Fighting développé et paru sur Super Nintendo en 1995. L'IA avait été programmée pour s'adapter à son adversaire en analysant ses coups et déterminer statistiquement les enchaînements les plus utilisés. Forte de ses connaissances, l'IA avait la capacité de prédire les coups adverses et les anticiper en bloquant ou en esquivant, difficile pour un joueur humain d'avoir les mêmes réflexes. Une des solutions pour rendre plus crédible ce type de comportement est d'intégrer des latences entre les décisions de l'IA.

Cette IA reste pertinente dans le cadre de l'apprentissage du joueur humain en haussant, peu à peu, son niveau de jeu. Elle a aussi un rôle à jouer pour améliorer le gameplay d'un jeu en faisant émerger des comportements de joueurs qui n'auraient pas été prévus par le game designer. Utile aussi donc dans une démarche d'amélioration de l'expérience de jeu.

L'IA en tant que joueur pour améliorer l'expérience de jeu est probablement l'une des plus utilisées. On la retrouve dans des jeux multijoueurs pour remplacer un joueur manquant dans une équipe. Aussi dans des modes solos destinés à le former aux mécaniques du jeu pour l'amener à un niveau qui lui permette d'être pleinement opérationnel dans les sessions multijoueurs.

Pour gérer les déplacements des PNJ, on use d'une IA qualifiée de bas niveau qui propose toute une panoplie d'algorithmes dédiés à la recherche de chemins (pathfinding) tels que Dijkstra ou A*. D'abord l'IA analyse l'environnement puis y déplace et positionne les PNJ en évitant des comportements idiots tel un PNJ qui essaierait de passer outre une porte fermée en s'y cognant sans arrêt qu'on a pu voir dans des jeux comme Fallout 4.

Le ridicule n'est pas que dans la navigation, il peut aussi apparaître dans le mouvement et la gestuelle avec des animations ou des effets sonores en contradiction avec l'action en cours. On lui donne de la crédibilité par adaptation de ses comportements et animations : face à un escalier, il doit monter et être animé en conséquence. En dehors des aspects purement comportementaux, l'esthétique des animations souvent réalisée grâce au motion capture, est aussi un sujet d'investigation de l'IA pour un rendu qui soit le plus naturel possible : le mouvement d'un personnage influencé par le port d'une charge, ou le sol sur lequel il se déplace.

Une IA pour donner aux PNJ un semblant de personnalité avec des comportements et des décisions qui lui sont propres et relatives à un environnement dynamique. Les PNJ sont amenés à poursuivre des objectifs cohérents avec la narration Pour gérer l'ensemble des comportements collectifs d'un groupe de PNJ que l'on retrouve notamment dans les jeux de sport collectif ou les jeux de stratégie, les modèles d'IA propose 2 approches distinctes dont l'une des clés est la capacité à analyser les actions du joueur.

La 1re consiste à gérer les PNJ avec des règles d'interaction entre eux et leur environnement. Et la 2de qui est un modèle déterministe planifie l'ensemble des comportements de chaque PNJ. Les solutions employées par les éditeurs sont souvent à mi-chemin des 2 solutions utilisant entre autres des automates à états finis ou encore des arbres de comportements. On retrouve ces approches, aussi simples soient-elles dans leur mise en œuvre, dès les années 70 dans des jeux comme Pong pour l'adversaire non humain ou Pacman pour donner un comportement à chaque fantôme. Par ce biais, le modèle assigne des objectifs priorisés et clairs à l'IA des PNJ. **Figure 1**

Unreal Engine met à disposition dans sa documentation toute une section dédiée à l'IA intégrant notamment des sections dédiées aux arbres de comportements, à la navigation, la collecte d'informations relatives à l'environnement et spatialisées par exemple la provenance d'un bruit ou encore la visibilité d'un élément. **Figure 2**

On utilise aussi l'IA pour permettre à chacun de jouer, quel que soit son niveau. Cela passe par l'adaptation dynamique de la narration et de la difficulté du jeu par apprentissage du comportement du joueur. Pour orienter la narration, on mémorise ce qu'il sait faire, ce qu'il a vu et ce qu'il connaît du jeu, l'histoire évolue en fonction des faits et gestes du joueur. Pour ce qui est de la difficulté, on mémorise ce qu'il sait et aime faire, sa capacité à faire pour lui proposer un gameplay adapté à son intérêt et à son niveau.

L'usage de l'IA a tendance à être dévoyé de son objectif initial relatif à l'amélioration de l'expérience de jeu en faveur d'objectifs plus mercantiles embarqués à la fois dans les modèles payants et free to play. Le jeu devient alors le média pour vendre par le biais de publicités bien placées et des achats intégrés allant de la facilitation de la progression du joueur (passage de niveaux, attribution de super armes, personnages aux compétences extraordinaires…) jusqu'à des achats totalement inutiles en dehors d'aspects purement esthétiques.

Un exemple d'intelligence artificielle

Simuler l'intelligence artificielle de PNJ n'est pas nécessairement complexe. Afin de la démystifier un peu, voici quelques règles d'interaction simples qui suffisent à concevoir une IA en capacité de gérer des comportements d'adversaires. La preuve avec ce petit jeu de course automobile sur papier facilement transposable en IA.

Munissez-vous d'une feuille de papier à petits carreaux et de crayons de couleur dont chacun sera attribué à un joueur. Une partie se fait avec au moins 2 joueurs.

La feuille de papier à carreaux est le support du plateau de

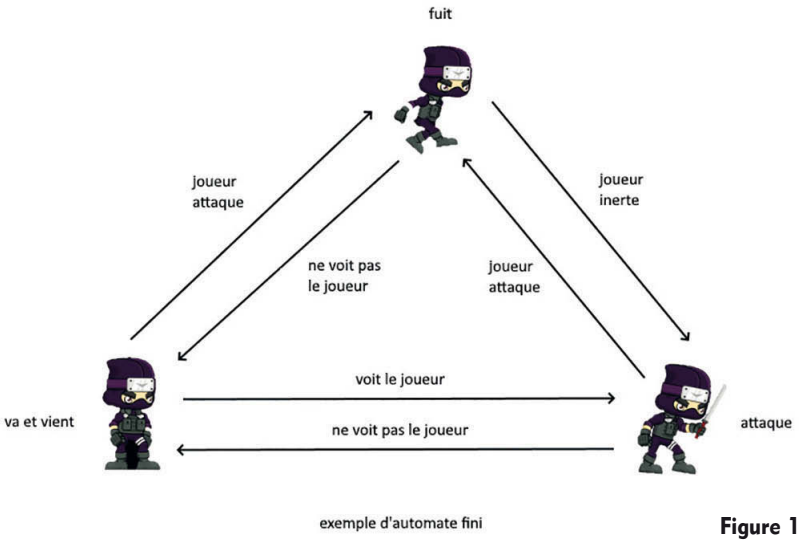


Figure 1

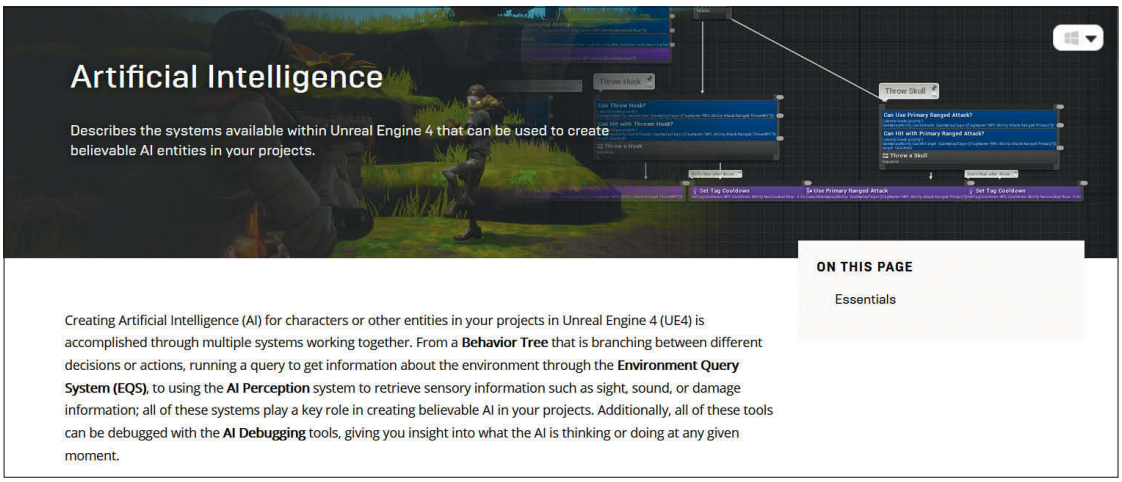


Figure 2

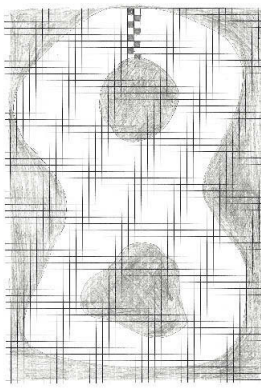
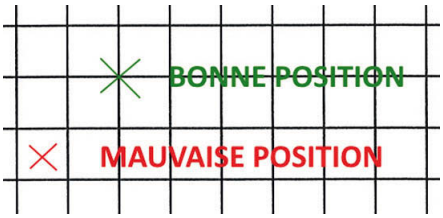


Figure 3

jeu qui doit représenter un circuit automobile dont vous avez la liberté de dessiner la forme ouverte ou fermée. Penser à ajouter une ligne de départ en damier. **Figure 3**

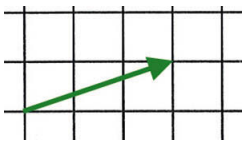
Les règles de base

- une voiture est toujours située sur l'intersection des lignes des carreaux, pas à l'intérieur du carreau.



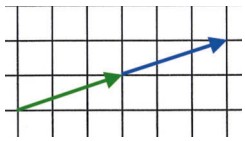
- on donne une direction au véhicule en traçant une ligne avec un stylo entre 2 intersections de carreaux ;
- on commence la partie avec un déplacement sur une case dans une des 4 directions nord, sud, est ou ouest ;
- les joueurs jouent chacun leur tour en commençant par le plus jeune ;
- le gagnant est le premier qui réussit à faire un tour de circuit ;
- à chaque tour, on déplace le véhicule du même nombre de cases dans la même direction que le coup précédent, soit d'une case de plus ou de moins dans l'une des 4 directions : nord, sud, est, ouest ;
- un déplacement d'un carreau au nord, sud, est ou ouest, permet au joueur de s'arrêter au coup suivant qui devient stationnaire. Au coup suivant, le déplacement sera limité à un carreau dans l'une des 4 directions.

Illustration des règles de base

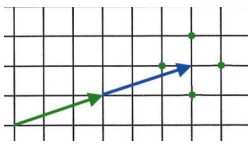


La flèche verte indique où vous vous êtes déplacé le tour précédent : 3 carreaux à l'est et un carreau au nord....

2 alternatives au coup suivant :

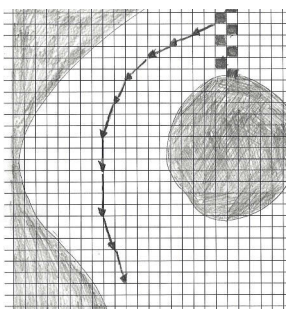


Ne rien faire ou suivre la flèche bleue dans la même direction que le coup précédent



Choisir de se déplacer sur un carreau supplémentaire au nord, au sud, à l'est ou à l'ouest sur l'un des points verts

...et au tour suivant se déplacer de 3 carreaux à l'est et choisir l'un des 4 points cardinaux



Dommages, crash et collisions.

Pas de collision délibérée, elles doivent être évitées. Et si cela se produit, pour chaque véhicule concerné, il y a retour à la position précédente. Au tour suivant, le déplacement sera limité à un carreau vers le nord, le sud, l'est ou l'ouest.

Chaque voiture subit des dommages correspondant au nombre de carreaux qui les séparaient avant l'accident.

En cas de choc avec un côté de la piste, le joueur subit un point de dégât pour chaque carreau qui le séparait du bord de la piste avant collision.

Pour redémarrer au tour suivant, la voiture est placée au bord de la piste sur le carreau le plus proche du choc avec un déplacement limité à un carreau vers le nord, le sud, l'est ou l'ouest.

Chaque voiture peut supporter 5 points de dommage. Au-delà, la partie est perdue.

Règles avancées

Une fois les règles de base maîtrisées après quelques parties, vous pouvez en intégrer de nouvelles plus complexes. En voici quelques exemples :

- Bonus Freins
  L'utilisation de bonus freins permet à la voiture de ralentir de 2 carreaux au lieu d'un. On peut en limiter l'usage à 2 fois par course du fait de l'usure.
- Turbo
  L'utilisation du « turbo boost » permet à la voiture d'accélérer de 2 carreaux au lieu d'un seul. Le « turbo boost » pourrait aussi être utilisable 2 fois par course.
- Flaque d'huile
  Dessinez des flaques d'huile sur le circuit avant de débuter la partie. Une voiture qui traverse une partie de la flaque pendant le virage, dérape et doit faire exactement le même déplacement que le coup précédent.

Customisation des bolides

Distribuez à chaque joueur la somme de 100€ pour s'équiper de bonus frein, de turbo ou de points de dégâts supplémentaires ou disposez ces bonus sur le circuit qui les octroie au joueur qui passe dessus.

Mode Mario Kart

Des bonus disséminés sur le circuit peuvent être récoltés par les joueurs qui passent dessus pour ensuite les utiliser.

Course en équipes

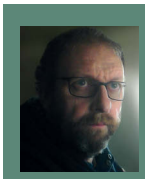
Formez des équipes de 2 ou 3 joueurs avec des règles du jeu inchangées. Les joueurs peuvent également conduire deux ou même trois voitures à la fois en les déplaçant chacune à chaque tour.

Trois points sont marqués pour une victoire, deux points pour une deuxième place et un point pour la troisième.

La seule limite : votre imagination avec des tas d'options possibles.

Créer un jeu avec Phaser.js

Phaser.js est un outil dont la prise en main est rapide. À l'opposé des artilleries lourdes que sont Unity, Godot, ou Unreal et dont l'utilisation est aussi simple et rapide que celle du tableau de bord d'un avion de ligne.



Franck Dubois
Video Game Codeur
Développeur agile
Formateur JavaScript / Unity / GDevelop

Avec ***Phaser.js***, on privilégie la simplicité avec pour seules armes, un éditeur comme Visual Studio Code (VS Code), un navigateur tel que Google Chrome et une (bonne) connexion internet. L'élégance étant de mise, nous monterons petit à petit, userons et abuserons avec ***Phaser.js*** de la norme ES6 qui rend la syntaxe et les concepts très proches de Java ou C#.

Configuration de l'environnement

Pas de temps à perdre non plus avec du matériel sophistiqué comme « node » ou « npm » qui pourrait ralentir la progression. Et pour gagner en célérité, on voyage léger en s'équipant dès à présent d'un navigateur web tel que Chrome et l'éditeur Visual Studio Code. Et si ce n'est pas déjà fait, installez-les.

Vous avez les armes, il ne manque plus que l'équipement pour faire vos tests qui n'est qu'un simple serveur http tel Apache ou Nginx installé par le biais de VS Code en suivant scrupuleusement les instructions suivantes.

Lancez « VS Code », puis depuis barre de menu située sur le bord gauche, cliquez sur l'icône  .

Saisissez les mots clés « Live Server » dans la zone de recherche, depuis la liste des extensions qui s'affichent, cliquez sur celle de Ritwick Dey et cliquez sur « Installer » pour l'ajouter. Testez le lancement de « Live Server » par le biais du raccourci « Ctrl+Shift+P » en sélectionnant « Live Server: Open with Live Server ». Le navigateur devrait se lancer en pointant sur l'adresse http://127.0.0.1:5500.

VS Code détectant toutes vos modifications sauvegardées, « Live Server » relance automatiquement le jeu : pas besoin donc de redémarrer « Live Server » à chaque modification.

Configuration du projet

Le projet du jeu vidéo prend naissance à partir de la création d'un premier dossier qui encapsulera tout le code source. Créez ce dossier depuis VS Code là où bon vous semble, par exemple le bureau, et nommez-le « shooter ».

Les applications ou sites web utilisent un fichier ***index.html*** à leur racine, fichier souvent appelé par défaut lorsque vous les lancez. Les jeux créés avec Phaser étant très proches du moins technologiquement parlant, n'échappent pas à la règle : ils utilisent aussi un fichier ***index.html*** appelé au lancement du jeu vidéo.

À la racine du projet, créez (clic droit + Nouveau Fichier) et éditez le fichier ***index.html***, et ajoutez-lui le code suivant :

```
<html>
 <head>
  <title>Shooter</title>
  <script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
</head>
<body>
 <script type="module" src="src/main.js"></script>
</body>
</html>
```

La ligne 4 permet d'importer dans le projet tous les fichiers nécessaires au bon fonctionnement de ***Phaser.js***. La ligne 7 fait référence à un fichier ***main.js*** placé dans le dossier ***src***. Il sera le point d'entrée du jeu qui permettra de lancer le jeu.

Pensez à sauvegarder après chaque modification pour qu'elles soient prises en compte par « Live Server ».

À la racine du projet, créez le dossier ***src***, puis dans ce même dossier, créez et éditez le fichier ***main.js*** et ajoutez-lui le classique suivant :

```
console.log('Hello, World!')
```

Testez la configuration du projet sur le navigateur en lançant les outils de développement web par le raccourci clavier « Ctrl+Shift+I ». Et vérifiez que la chaîne « Hello, World » s'affiche bien dans la console.

Phaser.js en JavaScript ES6

Dans un monde pas si lointain que ça, la syntaxe JavaScript ES6 était dépendante d'une multitude d'outils et librairies, un frein pour les débutants de l'époque qui nécessitait d'abord de bien connaître JavaScript.

Il nous reste 2 choses à faire pour terminer la configuration :

- transformer Phaser en module pour pouvoir l'exploiter avec ES6;
- configurer VS Code afin d'activer la complétion de code d'un projet Phaser.

Le terme module peut paraître un peu barbare pour le profane : pour faire simple, retenez qu'un module est à peu près la même chose qu'un fichier JavaScript et qu'il rassemble toute une série de fonctions connexes.

Pour importer un module, on utilise la commande « import » :

```
import { moduleObject } from './moduleName.js'
```

Pour qu'un objet puisse être exploité comme module, il doit au préalable être exporté via la commande « export » :

```
const moduleObject = {
 property1: 'property1Value',
 property2: property2Value
}

export {
 moduleObject
}
```

Phaser en module

Pour exploiter Phaser en tant que module, même principe. Tout d'abord, créez un sous-dossier ***lib*** dans le dossier ***src***.

Puis créez-y un fichier nommé ***phaser.js*** pour y mettre le code suivant servant à exporter ***Phaser*** en tant que module :

```
export default window.Phaser
```

Et pour importer ***Phaser*** en tant que module, remplacez le contenu du fichier ***src/main.js*** par le code suivant :

```
import Phaser from './lib/phaser.js'
console.dir(Phaser)
```

Testez la configuration du projet sur le navigateur en lançant les outils de développement web par le raccourci clavier « Ctrl+Shift+I » : vous devriez voir sous l'onglet console l'objet ***Phaser*** avec ses propriétés.

La complétion de code sous VS Code

La complétion est une aide très utile qui permet au développeur de lister les fonctions qu'il peut utiliser. En l'occurrence ici, les fonctions de la librairie ***phaser.js***.

Téléchargez le fichier ***phaser.d.ts*** depuis le lien sur programmez.com ou sur le GitHub du magazine.

Sous le dossier ***src***, créez un sous-dossier nommé ***types***, et placez-y le fichier ***phaser.d.ts***.

À la racine du projet, créez, éditez le fichier ***jsconfig.js*** et ajoutez le code suivant :

```
{
 "compilerOptions": {
 "module": "es6",
 "target": "es6"
 }
}
```

Vérifiez la complétion en tapant dans le fichier ***main.js*** le mot clé Phaser suivi d'un point « . », VS Code devrait pour proposer une liste d'instructions propres à Phaser.

L'environnement de développement est désormais prêt pour commencer à coder le « shooter ».

Création du jeu vidéo

Le jeu vidéo développé ici est un shoot'em up que nous appelons « shooter », un jeu d'arcade à scrolling vertical. Un des classiques des jeux depuis les années 1980. Cet exemple permet de travailler les 3 piliers composants la boucle du jeu ou de gameplay que sont l'objectif, le défi et la récompense.

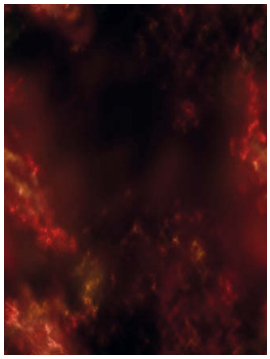
Les différentes étapes :

- un fond d'écran, qui défile (scrolle) du haut vers le bas, en continu et à l'infini pour donner une impression de mouvement;
- un shooter contrôlé par le joueur ;
- un laser lancé par le shooter à l'initiative du joueur ;
- des météorites qui fusent de partout ;
- des effets sonores ;
- un score ;
- un game over.

Les assets

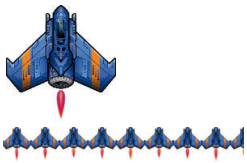
En premier lieu, téléchargez depuis le lien suivant les images ci-dessous. Vous y trouverez aussi les effets sonores du jeu.

Téléchargez le dossier assets et copiez-le à la racine du projet.



Le fond d'écran à scroller
(480x600 pixels)

Le laser (PJ)



Le spaceship dirigé par le joueur (PJ)

Une météorite (PNJ)

Une explosion (PNJ)

Où trouver des assets ?

Nombreux sont les sites web qui proposent des assets gratuits ou payants. Les droits d'usage variables peuvent être liés à certaines obligations telles que citer les auteurs et ajouter un lien depuis votre jeu. Pour les animations, je vous invite à regarder du côté de opengameart, itch.io et craftpix. Pour les effets sonores, regardez du côté de digccmixter, encore itchi.io ou universal soundbank.

Initialisation du jeu

L'initialisation commence par la création d'une instance Phaser depuis le fichier ***main.js*** dans lequel on indique :

- le mode de rendu (canvas ou WebGL) en renseignant la propriété ***type***;
- et la résolution en renseignant les propriétés ***width*** et ***height***.

```
export default new Phaser.Game({
 type: Phaser.AUTO,
 width : 600,
 height : 800
})
```

Le mode de rendu spécifié ici est ***Phaser.AUTO*** laissant à Phaser le choix d'adopter le rendu le plus pertinent selon le navigateur ou la machine d'exécution.

Création de la première scène

Un jeu vidéo comporte différentes scènes. Pour faire simple, chaque écran d'un jeu vidéo est une scène : une cinématique, un menu, un niveau, les meilleurs scores, etc.

Sous Phaser comme beaucoup d'autres, un jeu vidéo est donc constitué de scènes indépendantes les unes des autres : elles peuvent être travaillées individuellement les unes des autres. Elles sont ensuite reliées, tout comme sont montés les films au cinéma, pour donner naissance au jeu dans sa globalité. Les scènes, par nature, n'ont pas vocation à coexister à un même moment lors de l'exécution du jeu.

Le « shooter » ne va comporter qu'une seule scène incluant introduction, game over et jeu. Pour créer une scène sous Phaser, il suffit de créer une classe qui étend la classe ***Phaser.Scene***.

Commencez par créer sous ***src*** un sous-dossier ***scenes*** dédié, puis le fichier de la classe ***Game*** que vous nommez ***Game.js***,

dans lequel vous importez la classe ***Phaser*** depuis le fichier ***phaser.js***. et définissez la scène Game en tant que fille de ***Phaser.Scene***.

```
import Phaser from '../lib/phaser.js'

export default class Game extends Phaser.Scene {
 constructor() {
  super('game') ;
 }
 preload() {
 }
 create() {
 }
 update() {
 }
}
```

Ce code comporte 4 méthodes :

- ***constructor*** : chaque scène doit être identifiée par une clé unique que vous définissez en paramètre de la fonction ***super*** : la clé choisie ici est ***game*** ;
- ***preload*** : c'est depuis cette fonction que sont chargés les assets : ce chargement préalable est incontournable, et sans cela impossible de les intégrer au jeu ;
- ***create*** : ici on crée la forme de la scène à son démarrage avec tous les objets nécessaires (préalablement définis dans la fonction preload);
- ***update*** : cette méthode est appelée en continue, les opérations donnant vie au jeu sont donc à placer ici.

Une fois la scène créée, on la rattache au jeu lors de la création de l'instance Phaser par le biais de son nom (***Game***) dans la propriété ***Scene*** et on importe la classe de la scène ***Game*** dans le projet. Le jeu ne nécessitant pas d'interactions complexes entre objets on spécifie le moteur physique ***arcade***. auquel on peut activer le mode ***debug*** qui ajoute quelques artifices d'affichage bien utiles.

```
import Phaser from './lib/phaser.js'
import Game from './scenes/Game.js'

export default new Phaser.Game({
 type : Phaser.AUTO,
 width : 600,
 height : 800,
 scene : Game,
 physics : {
  default : 'arcade',
  arcade : {
   debug : true
  }
 }
})
```

Le fond d'écran et le scrolling de la scène de jeu

La 1re chose à faire est d'intégrer et d'afficher le fond d'écran ***nebula.red.png*** qui se fait en 2 temps :

- le chargement de l'image depuis le fonction ***preload*** ;
- l'affichage de l'image (rattachée à la propriété ***background*** de la scène) au centre de la scène depuis la fonction ***create***.

```
preload() {
 // chargement de l'image en lui associant un identifiant appelé 'background'
 this.load.image('background', 'assets/background/nebula.red.png');
}
create() {
 // ajouter l'image au centre de la scène : utilisation de son identifiant
 'background'
 this.background = this.add.image(300, 400, 'background');
}
```

Notez que les objets sont toujours placés sur scène à partir de leur centre. En l'état, le fond d'écran s'affiche. Pour le faire scroller, un simple déplacement vers le bas en incrémentant son ordonnée suffit. C'est ici que la méthode update intervient popur y implémenter le changement de position du fond d'écran. **Figure 1**

```
import Phaser from '../lib/phaser.js'

export default class Game extends Phaser.Scene {
 .....
 update() {
  // déplacer le « background » vers le bas
  this.background.y += 1;
 }
}
```

Oui, mais il y a un os, ça scrolle bien, mais pas de manière continue : le fond d'écran disparaît. Pour avoir un scrolling continu, il ne faut pas faire scroller un fond d'écran, mais 2 en les juxtaposant l'un sur l'autre. **Figure 2**

Et aussi faire en sorte que lorsque l'un des ***background*** sort de la scène, il soit replacé juste au-dessus de l'écran de jeu. Cela nécessite d'ajouter le 2d « background » à la méthode ***create***, de le déplacer depuis la méthode ***update***, et dans cette même méthode replacer les 2 « background » au sortir de l'écran de jeu.

Pour optimiser, on intègre les 2 ***background*** dans un groupe d'objets statiques puis pour chacun de ses éléments, on le replace sur leur position d'origine s'ils sont sortis de l'écran de jeu.

```
create() {
 this.backgrounds = this.physics.add.staticGroup();
 this.backgrounds.create(300, 400, 'background');
 this.backgrounds.create(300, -400, 'background');
}

update() {
 this.backgrounds.children.iterate(child => {
  const background = child;
  background.y += 1;
  if ( background.y > 1200 )
   background.y = -399
 })
}
```

Il y a encore plus simple grâce à l'objet ***TileSprite***, objet se comportant comme un tapis roulant : une image qui s'enroule et que l'on peut déplacer à l'infini. C'est l'objet par excellence utilisé pour créer des scrollings d'arrière-plan bien que l'usage des caméras puisse être une solution.

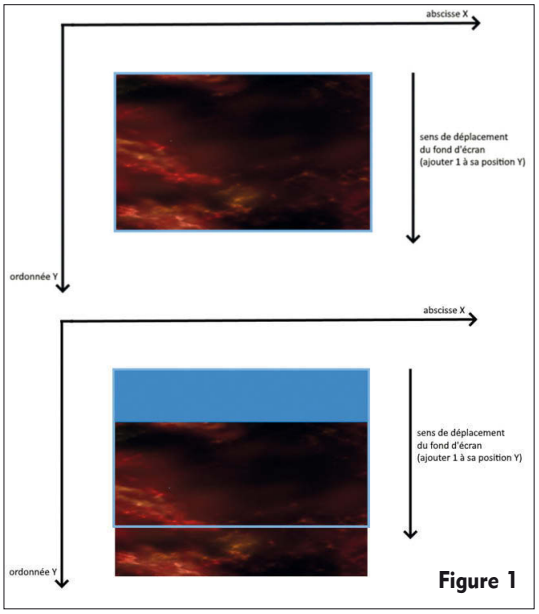


Figure 1

```
create() {....
 this.background = this.add.tileSprite(300,400,600,800,'background');
....}

update() {....
 this.background.tilePositionY -= 5;
....}
```

Le shooter

Le joueur dont l'objectif est de se frayer un chemin dans un champ de météorites se trouve ici représenté par l'image ***shooter.png*** : référencée dans la fonction ***preload*** et ajoutée à la scène depuis la fonction ***create***.

```
preload() { ....
 this.shooter = this.load.image('shooter', 'assets/shooter/shooter.png');
}
create() { ....
 this.physics.add.sprite(300, 600, 'shooter');
}
```

L'animation du shooter

Pour réaliser l'animation, on utilise la planche de sprites ***tildShooter.png*** référencée en ***preload*** en tant que ***sprite-sheet*** en lui spécifiant en propriétés la taille d'une image. En ***create*** vient la configuration de l'animation à laquelle on donne un nom (***key***), une vitesse (***framerate***), la liste des images la composant (***frames***) et le nombre de fois que l'animation est jouée (***repeat***) dont une valeur positionnée à ***-1*** engendre un jeu en boucle.

```
preload() { ....
 this.load.spritesheet("shooter", "assets/shooter/tiledShooter.png", {
frameWidth: 53.5, frameHeight: 60 });
}

create() {.....
 this.anims.create({
  key: "fly",
  frameRate: 30,
  frames: this.anims.generateFrameNumbers("shooter", { start: 0, end:
 19 }),
```

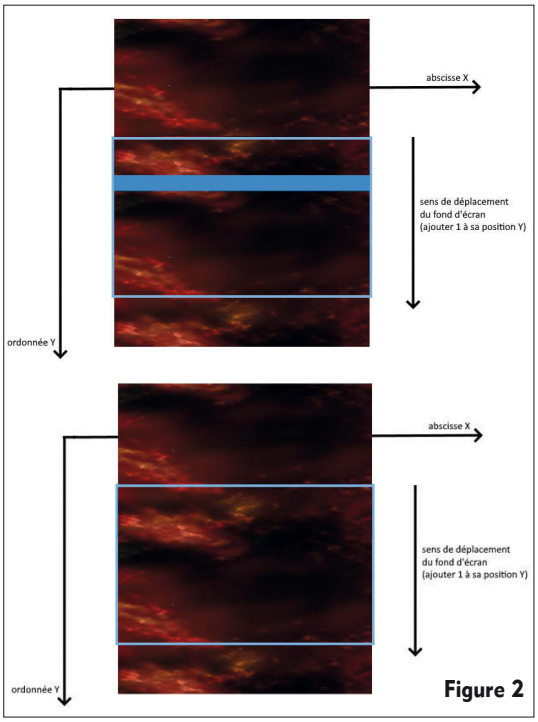


Figure 2

```
  repeat: -1
 });
}

 this.shooter = this.physics.add.sprite(300, 600, 'shooter');
 this.shooter.play("fly");
}
```

Contrôle et déplacement du shooter

On donne au joueur la faculté de contrôler le shooter à l'aide des flèches directionnelles : on ajoute une propriété ***cursors*** chargée de gérer les touches du clavier qu'on initialise dans ***create***. Puis on intègre les déplacements du shooter associés aux touches depuis ***update*** en lui donnant de la vitesse.

Code complet sur programmez.com & github

Blocage du shooter sur les bords de la scène

Il suffit juste d'activer depuis ***create*** le rebond des objets sur les bords de la scène. La méthode ***this.physics.world .setBoundsCollision*** peut prendre 4 paramètres booléen correspondant aux 4 côtés de la scène dont on peut activer ou désactiver le rebond sur chacun et dans l'ordre gauche, droite, haut et bas.

```
create() {....
 this.physics.world.setBoundsCollision();
....}
```

Et de stopper le shooter lorsqu'il entre en collision avec l'un des 4.

```
create() {.....
 this.shooter = this.physics.add.sprite(300, 600, 'shooter');
 this.shooter.body.collideWorldBounds = true;
 this.shooter.play("fly");
}
```

Contrôle du laser par le joueur

Pour éviter de finir avec du code spaghetti et en y allant crescendo, on instille de la modularité avec l'objet ***Laser*** en tant

que ***Sprite***. Libre à vous de l'adapter aux objets ***background***, et ***shooter***.
Retenez que d'un point de vue architecture et comme ailleurs, il y a des effets de mode : il en existe à foison qui ont chacune leurs promoteurs et détracteurs. L'une d'entre elles est le modèle ECS (Entité Composant Système) plébiscité dans la création de jeux vidéo. Parfois intégrée au moteur, elle est plus ou moins facile à implémenter et utiliser sur certains pas nativement conçus pour.
Pour des interactions réalistes « conformes » à la physique, il est possible de spécifier certaines propriétés aux objets ***Sprite*** telles que la masse, la sensibilité à la gravité ou jouer sur l'inertie avec les accélération et décélération.
Le code de l'objet est encapsulé dans un fichier dédié ***Laser.js*** placé sous le dossier ***scene***. Pour que cette classe soit accessible, il est nécessaire de l'importer depuis la scène définie dans le fichier ***Game.js***.

```
import Laser from './Laser.js'
```

Cet objet prend en charge la définition des animations qui lui sont rattachées, son propre rattachement à une scène ainsi qu'une méthode lui donnant le mouvement tout en l'animant.
Au préalable, on charge en mémoire la planche des animations du laser depuis ***preload***.

```
preload() {....
 this.load.spritesheet('laser', 'assets/laser/tiledLaser.png', { frameWidth: 20,
 frameHeight: 64 });
....}
```

Le constructeur ***constructor*** associe la scène à l'objet, définit les animations et prend 4 paramètres :
- la scène sur laquelle l'objet sera placé ;
- les positions x et y de l'objet sur la scène ;
- la vitesse du laser.

```
export default class Laser extends Phaser.Physics.Arcade.Sprite {

 constructor(scene, x, y, velocity) {
  super(scene,x,y,'laser');
  this.velocity = velocity ;
  scene.anims.create({
   key: 'run',
   frameRate: 60,
   frames: scene.anims.generateFrameNumbers('laser' , { start: 7, end: 0 }),
   repeat: 1
  });
  scene.add.existing(this);
  scene.physics.add.existing(this);
 }

 start() {
  this.play('run');
  this.setVelocityY(this.velocity);
 }
}
```

Le lancement du laser se fait à l'initiative du joueur en appuyant sur la touche Espace en faisant en sorte qu'un seul laser à la fois soit lancé. La méthode ***start*** qui joue l'animation, applique une vitesse de déplacement verticale vers le haut est alors appelée.

```
update() {.....
 // player control
 if ( this.cursors.left.isDown) {
  this.shooter.setVelocityX(-200);
 } else if ( this.cursors.right.isDown) {
  this.shooter.setVelocityX(200);
 } else if ( this.cursors.up.isDown) {
  this.shooter.setVelocityY(-200);
 } else if ( this.cursors.down.isDown) {
  this.shooter.setVelocityY(200);
 } else if ( this.cursors.space.isDown ) {
  this.readyToUp = true;
 } else if ( this.readyToUp && this.cursors.space.isUp ) {
  this.laser = new Laser(this, this.shooter.x, this.shooter.y-this.shooter.
height, -300);
  this.laser.start();
  this.readyToUp = false;
 } else {
  this.shooter.setVelocity(0);
 }
}
```

La météorite et son mouvement

Le champ de météorites est composé sans surprise de météorites, plus ou moins grandes, plus ou moins rapides avec des trajectoires plus ou moins chaotiques.
Au départ et pour faire simple, on donne à la météorite un mouvement vertical du haut vers le bas de la scène, en lui appliquant une vitesse sur l'axe des ordonnées. La méthode d'animation diffère de celle du shooter puisqu'ici elle utilise une simple rotation du sprite sur lui-même.
Notez que la trajectoire de la météorite est un moyen très simple et peu coûteux pour accroître la difficulté. La synthèse des différents leviers à travailler peut se faire à l'aide d'un simple tableau. En jouant sur les curseurs de chacun, il devient facile de durcir le challenge et d'ajuster la courbe de difficulté comme vous pouvez en voir une illustration sommaire ci-dessous.

Levier / Niveau	*	**	***	****
Nombre lancés	25	50	100	150
Trajectoire	Rectiligne verticale	Rectiligne verticale	Rectiligne verticale et horizontale	Rectiligne verticale, horizontale et de biais
Vitesse	Uniforme	Croissante	Croissante	Croissante
Taille	Uniforme	Uniforme	Croissante	Aléatoire

Sur le même modèle que celui du ***Laser***, on crée un objet ***Meteor***. Avec un constructeur, et 2 méthodes ***start*** pour lancer la météorite du haut vers le bas et ***anim*** pour l'animer en lui appliquant une rotation donnée en degrés par seconde.

```
export default class Meteor extends Phaser.Physics.Arcade.Sprite {

 constructor(scene, x, y, texture) {
  super(scene,x,y,texture);
  scene.add.existing(this);
 }

 start(velocityX,velocityY) {
```

```
 this.setVelocityX(velocityX);
 this.setVelocityY(velocityY);
}

 anim(angularVelocity) {
  this.setAngularVelocity(angularVelocity) ;
 }
}
```

Idem que pour le laser, le code de l'objet est encapsulé dans un fichier dédié ***Meteor.js*** placé sous le dossier ***scene***. Pour que cette classe soit accessible, il est nécessaire de l'importer depuis la scène définie dans le fichier ***Game.js***. et de charger les 2 images des météorites.

```
preload() { ...
 this.load.image('meteor', 'assets/meteor/meteor.png');
 this.load.image('meteor2', 'assets/meteor/meteor2.png');
}
```

Les météorites et leurs mouvements

Pour rendre le challenge à minima intéressant, ce n'est pas une mais 50 météorites qui seront lancées séparément à des endroits choisis au hasard sur l'horizontale. L'utilisation d'un ***timer*** est particulièrement adaptée : son paramétrage permet de lancer des actions à intervalle régulier en en limitant le nombre. ***Phaser.js*** intègre plusieurs structures de données qui proposent de rassembler des objets par nature et connexes dans leur comportement.

La structure s***taticGroup*** regroupe des objets inertes ***StaticBody***, qui nécessitent une gestion manuelle des interactions et la structure ***group*** regroupe des objets ***Body*** qui subissent les affres de l'environnement dans lequel ils évoluent, comme la gravité ou d'autres objets en faisant partie. Environnement, par ailleurs, configurable par le biais de la propriété ***physics*** de l'objet ***Phaser.Game***.

Nos objets étant animés et entrant en interaction les uns avec les autres, on crée un ***group*** en spécifiant les classes d'objets qui y seront rattachés depuis ***create***.

Par soucis d'équité entre les joueurs, le déroulé d'une partie doit toujours être le même : d'une partie à l'autre, les météorites suivent toujours le même ordre et le même chemin. Positionner une météorite par le simple tirage d'un nombre aléatoire ne remplit pas cette mission puisque la suite de nombres va varier d'une partie à l'autre. Pour éviter cet écueil, on utilise une génération contrôlée de nombre aléatoires : la suite de nombres sera toujours la même.

Phaser.js a dans sa panoplie d'objets, un ***RandomData Generator*** qui prend en paramètre un nombre appelé graine (***seed***). Pour une ***seed*** donnée, la suite des nombres tirés au sort est toujours la même. Cette approche est aussi exploitée dans la génération procédurale d'espaces par le biais de l'algorithme de Perlin ou dérivés. On initialise ce générateur avec la graine 999 depuis ***create***.

```
create() {....
 this.randomData = new Phaser.Math.RandomDataGenerator(999);
 this.meteorGroup = this.physics.add.group({
  classType: Meteor
 });
 .....}
```

La configuration du type d'objets via ***classType*** du groupe permet au groupe de créer des objets de ce même type. On ajoute une méthode ***spawnMeteor*** dédiée à la création des météorites par le biais de ***meteorGroup*** qui prend en paramètres sa forme, les vitesses sur les 2 axes, la vitesse de rotation et la scène à laquelle se rattacher.

```
spawnMeteor(meteorView, velocityX, velocityY,angularVelocity,scene) {
 const meteor = this.meteorGroup.get(this.randomData.between(0, 600),
50,meteorView);
 scene.add.existing(meteor);
 meteor.start(velocityX, velocityY);
 meteor.anim(angularVelocity);
}
```

Il ne reste plus qu'à générer chaque seconde les météores avec le ***timer*** en choisissant au hasard leur image.

```
create() {....
 this.meteorTimer = this.time.addEvent( {
  delay : 1000,
  callback: () => {
   let meteorRandomView =  Math.ceil(Math.random()*2);
   let meteorView = 'meteor';
   if ( meteorRandomView == 1 ) {
    meteorView = 'meteor2';
   }
   this.spawnMeteor(meteorView,0,100,15,this) ;
  },
  loop: false, repeat : 50, callbackScope: this
 });
....}
```

Pour jouer sur la difficulté, il suffit d'activer les leviers sur la vitesse, la trajectoire ou la taille de l'objet ***Meteor***.

Les collisions

Les collisions peuvent être gérées manuellement ou par le biais du moteur physique via la propriété ***body*** d'un sprite. Dans ce cas, la gestion d'une collision est simplifiée et se fait par des « ***collider*** » pour les collisions ou des « ***overlap*** » pour les chevauchements respectivement par les méthodes « ***scene.physics.add.collider*** » ou « ***scene.physics.add.overlap*** ». La différence étant qu'avec un « ***collider*** » et du fait des déplacements et des forces, les trajectoires des objets s'en voient modifiées.

On s'occupe du cas du shooter et des météorites : si la ***météorite*** percute le ***shooter***, elle explose.

```
preload() { ...
 this.load.spritesheet('meteorExplosion', 'assets/tiledExplosion.png', {
frameWidth: 128, frameHeight: 64 });
}
```

```
create() {....
 this.physics.add.overlap(this.shooter,this.meteorGroup,this.handleShooter
MeteorCollision,undefined,this);
....}
```

L'instruction ci-dessus faite depuis « ***create*** », génère un appel à la fonction « ***handleShooterMeteorOverlap*** » lorsqu'il y a chevauchement entre le ***shooter*** et une ***météorite***.

```
handleShooterMeteorCollision(shooter,meteor) {
 meteor.play('explosion');
 meteor.once(Phaser.Animations.Events.ANIMATION_COMPLETE, () => {
  meteor.destroy();
 });
}
```

On définit l'animation de l'explosion depuis l'objet ***Meteor***.

```
export default class Meteor extends Phaser.Physics.Arcade.Sprite {

 constructor(scene, x, y, texture) {
  super(scene,x,y,texture);
  scene.anims.create({
   key: 'explosion',
   frameRate: 25,
   frames: scene.anims.generateFrameNumbers('meteorExplosion', { start:
 0, end: 10 }),
   repeat: 0
  });
}
```

« ***handleShooterMeteorCollision*** » joue l'animation de l'explosion et lorsqu'elle est jouée, la météorite est détruite. En l'état, c'est imparfait puisque l'animation de l'explosion est bloquée tant que le ***shooter*** chevauche la ***météorite*** et tourne sur elle-même. Pour éviter cet écueil, il suffit de désactiver le moteur physique de l'objet en appelant la méthode ***disableBody***.

```
handleShooterMeteorCollision (shooter,meteor) {
 meteor.disableBody();
 meteor.play('explosion');
 meteor.once(Phaser.Animations.Events.ANIMATION_COMPLETE, () => {
  meteor.destroy();
 }) ;
}
```

On utilise le même principe pour les collisions entre le laser et les météorites. Toutefois, la création de l'objet ***Laser*** avec l'opérateur ***new*** nous oblige à définir un test ***overlap*** à chaque fois.

Il apparaît donc plus pertinent de faire comme pour les météorites en créant un groupe dynamique qui créera les lasers ainsi qu'un test ***overlap*** unique entre les ***lasers*** et les ***météorites*** avec un appel à une fonction callback « ***handleLaserMeteorCollision*** » qui pour le moment, est identique à « ***handleShooterMeteorCollision*** ».

```
create() {....
 this.laserGroup = this.physics.add.group({
  classType: Laser
 });
 ....
 this.physics.add.overlap(this.laserGroup, this.meteorGroup,this.handle
LaserMeteorCollision,undefined,this);
....}

update() {....
 } else if ( this.readyToUp && this.cursors.space.isUp ) {
  this.laser = this.laserGroup.get(this.shooter.x, this.shooter.y-this.shooter.
height,'laser');
  this.add.existing(this.laser);
  this.laser.velocity = -300;
  this.laser.start();
  this.readyToUp = false;
 } else {
....}
```

Calque et layer

Jusqu'à présent, on gérait des objets de même nature parce qu'interagissant les uns avec les autres. Afin de travailler sur des aspects du jeu de nature différente, une bonne pratique est de les regrouper dans des feuilles virtuelles appelées calques (layer) que l'on pourra afficher ou cacher à la demande et ordonner les uns par rapport aux autres.

On crée 2 calques : un pour les objets graphiques animés, l'autre pour les informations relatives au jeu telles que la barre de vie ou plus tard le score que l'on place au premier plan avec la méthode ***setDepth***.

```
create() {....
 this.playerInfLayer = this.add.layer();
 this.gameLayer = this.add.layer();
 this.playerInfLayer.setDepth(2);
 this.gameLayer.setDepth(1);
....}
```

On rattache shooter, background, laser et météorite au layer ***playerInfLayer*** au moment de leur création.

```
this.gameLayer.add([this.background,this.shooter]);
```

```
} else if ( this.cursors.space.isUp && this.readyToUp ) {
 this.laser = this.laserGroup.get(this.shooter.x, this.shooter.y-this.shooter.
height,'laser');
 this.add.existing(this.laser);
 this.gameLayer.add(this.laser);
 ....
```

On change la signature de la méthode ***spawnMeteor*** afin d'y intégrer le ***layer*** en paramètre.

```
spawnMeteor(meteorView, velocityX, velocityY,scene, layer) {....
 layer.add(meteor);
 ....}
```

Sans oublier de modifier aussi son appel par le biais du timer.

```
this.meteorTimer = this.time.addEvent( {....
 this.spawnMeteor(meteorView,0, 100,this,this.gameLayer);
....},
```

La barre de vie du shooter

C'est par le biais des fonctions graphiques de phaser.js qu'on dessine la barre de vie qui passe du vert au rouge en dessous de 30%.

Un nouvel objet ***HealthBar*** (***HeathBar.js***) dédié qui hérite de la classe ***Phaser.GameObjects.Graphics*** et qui définit un constructeur, 2 méthodes pour décrémenter la barre de vie et l'autre pour la dessiner.

4 propriétés pour ***HealthBar*** : sa position x et y, sa valeur absolue, et un ratio pour adapter sa taille à celle de la scène.

3 méthodes : le constructeur prenant en paramètre la scène à laquelle il est rattaché et sa position, une méthode pour diminuer la taille de la barre de vie appelée lors des collisions entre le shooter et les météorites et une autre pour la dessiner.

Code complet sur programmez.com & github

Il ne reste plus qu'à intégrer la barre de vie en bas à gauche de la scène depuis ***create*** tout en l'intégrant au ***layer*** puis d'appeler sa méthode ***decrease*** depuis la fonction « ***handleShooterMeteorCollision*** ».

```
...
import HealthBar from './HealthBar.js'
....
create() {....
 this.healthBar = new HealthBar(this, 0, 392);
 this.playerInfLayer.add(this.healthBar);
....}
....
handleShooterMeteorCollision(shooter,meteor) {....
 this.healthBar.decrease(10);
....}
```

Les effets sonores

On fait simple en intégrant 2 effets sonores : le 1er au lancement du laser et le 2d lorsque les météorites explosent.

Tout comme les sprites et animations, avant de pouvoir jouer tout effet sonore, il faut au préalable les charger en mémoire depuis la méthode ***preload***.

```
preload() {....
 this.load.audio('laser', 'assets/audio/laserSound.mp3');
 this.load.audio('explosion', 'assets/audio/explosion.wav');
....}
```

On intègre à la scène 2 nouvelles propriétés ***laserSound*** et ***explosionSound*** au moment de sa création depuis ***create***.

```
create() {....
 this.laserSound = this.sound.add("laser");
 this.explosionSound = this.sound.add("explosion");
....}
```

En appelant sur ***laserSound*** et ***explosionSound*** la méthode ***play***, on joue le son du laser à son lancement.

```
update() {....
 } else if ( this.readyToUp && this.cursors.space.isUp ) {
  this.laser = this.laserGroup.get(this.shooter.x, this.shooter.y-this.shooter.
height,'laser');
  this.add.existing(this.laser);
  this.laserSound.play() ;
  this.gameLayer.add(this.laser);
  this.laser.velocity = -300;
  this.laser.start();
  this.readyToUp = false;
 } else {
....}
```

Et l'explosion lors des collisions entre laser ou shooter.

```
handleLaserMeteorCollision(shooter,meteor) {
 meteor.disableBody();
 this.explosionSound.play();
....}

handleShooterMeteorCollision(shooter,meteor) {
 meteor.disableBody();
 this.explosionSound.play();
....}
```

Le score

Pour afficher le score, on a tout d'abord besoin d'ajouter une nouvelle variable nommée ***score*** initialisée à 0.

```
export default class Game extends Phaser.Scene
{
 score = 0;
....}
```

Puisque la règle est de 10 points gagnés lorsqu'un laser touche une météorite, on ajoute 10 à la variable score depuis la méthode « ***handleLaserMeteorCollision*** ».

```
handleLaserMeteorCollision(shooter,meteor) {....
 this.score += 10;
....}
```

Il reste à afficher ce score sur la scène dans le bon calque, d'abord au lancement du jeu depuis ***create***. On utilise un objet ***Text*** qui prend en paramètres sa position, son contenu et sa forme au format ***css***.

```
create() {....
 const style = { color: '#ffffff', fontSize: 1, fontFamily: 'Verdana' };
 this.scoreText = this.add.text(0, 0, 'SCORE : 0', style);
 ....
 this.playerInfLayer.add([this.scoreText,this.healthBar]);
....}
```

Puis lorsqu'il change en faisant une interpolation en le plaçant juste après la chaîne de caractères « Score : ».

```
handleLaserMeteorCollision(shooter,meteor) {....
 this.score+=10;
 const value = `SCORE: ${this.score}`;
 this.scoreText.text = value;
....}
```

L'instruction ***this.scoreText.text*** va remplacer ***${this.score}*** par la variable ***score***.

Le game over et nouvelle partie

Afin de stopper toutes les animations et interactions, on ajouter une propriété ***gameOn*** à la scène fixée à **false**. Et on encapsule tout le contenu de la méthode ***update*** dans un test sur cette nouvelle variable.

```
export default class Game extends Phaser.Scene
{
 gameOn = false;
 ....
 update() {
  if ( this.gameOn ) {
   ....
  }
}
```

Avec 2 images, on choisit d'afficher un « game over » lorsque la barre de vie du joueur est épuisée et un « play » pour relancer la partie. Cela devient une habitude maintenant : on les charge depuis ***preload***.

```
preload() {....
 this.load.image('play', 'assets/play.png');
 this.load.image('gameOver', 'assets/gameOver.png');
....}
```

Depuis ***create***, on les ajoute à la scène et au layer ***playerInfLayer*** en le centrant et en laissant visible l'image ***play***.

```
create() {....
 this.play = this.add.image(this.sys.game.canvas.width/2,this.sys.game.
canvas.height/3,'play');
 this.play.setVisible(true);
 this.gameOver = this.add.image(this.sys.game.canvas.width/2,this.sys.game.
canvas.height/1.5,'gameOver');
 this.gameOver.setVisible(false);
 ....
 this.playerInfLayer.add([this.scoreText,this.healthBar,this.play,this.game
Over]);
...}
```

On les rend visibles lorsque la barre de vie du joueur est vide tout en supprimant les ***météorites***, en cachant le ***shooter*** et en fixant la valeur de ***gameOn*** à ***false***.

```
handleShooterMeteorCollision(shooter,meteor) {....
 if ( this.healthBar.decrease(10) ) {
  this.gameOver.setVisible(true);
  this.play.setVisible(true);
  this.shooter.setVisible(false);
  this.meteorGroup.clear(true,true);
  this.gameOn = false;
 }
....}
```

On active le ***spawn*** des météorites uniquement lorsqu'une partie est en cours.

Code complet sur programmez.com & github

Pour relancer une partie, il suffit de cliquer sur l'image ***play***, et de remettre les compteurs à 0 : le score, la barre de vie, la génération des météorites, et rendre visible le shooter.

On commence par rendre sensible au clic l'image ***play*** avec la méthode ***setInteractive*** puis on traite l'événement clic rattaché duquel on appelle toutes les instructions nécessaires à la remise en marche du jeu.

Code complet sur programmez.com & github

Les interactions physiques

Pour accroître la difficulté, on donne aux météorites des trajectoires qui ne soient ni verticales ni horizontales et fait en sorte que lorsque 2 d'entre elles se percutent leurs trajectoires respectives soient affectées.

Toujours pour que le déroulement du jeu soit uniforme d'un joueur à l'autre, depuis ***create*** on définit un 2^d ***RandomDataGenerator*** pour tirer au hasard et de manière ordonnée les vitesses des météorites sur chacun des 2 axes au sein de la méthode ***spawnMeteor***, dont on va une nouvelle fois changer la signature en supprimant les paramètres de vitesse.

```
create() {....
 this.randomDataMeteor = new Phaser.Math.RandomDataGenerator(999);
....}
```

```
....
spawnMeteor(meteorView, angularVelocity, scene, layer) {....
 meteor.start(this.randomDataMeteor.between(-200, 200),this.random
DataMeteor.between(20, 300));
....}
```

Sans oublier de modifier aussi son appel par le biais du ***timer***.

```
this.meteorTimer = this.time.addEvent( {....
 this.spawnMeteor(meteorView,15,this,this.gameLayer);
....},
```

En l'état, les trajectoires sont bien variées, mais les collisions restent sans effet. Souvenez-vous des ***overlap*** et des ***collider*** dont nous avions parlés précédemment : ici pour qu'il y ait interaction on définit un ***collider*** entre le groupe ***meteorGroup*** et lui-même toujours au sein de ***create***. Et il n'y a que ça à faire.

```
create() {....
 this.physics.add.collider(this.meteorGroup, this.meteorGroup);
....}
```

Tout marche à une imperfection près : les interactions ne sont pas des plus précises. Les objets étant par défaut affublés d'un masque de collision rectangulaire, les changements de trajectoire se font en conséquence. ***Phaser.js*** permet de gérer des masques de collision complexes.

Nonobstant, on fait un choix plus simple : les météorites étant à peu près circulaires, l'usage d'un masque du même acabit suffit à donner plus de réalisme. On le spécifie au sein de la fonction ***spawnMeteor*** par appelant la méthode ***setCircle*** qui prend en paramètre le rayon du masque en pixels. Les météorites faisant 64 pixels de côté, on passe la valeur 32. Il reste à déterminer la restitution ou la transmission des forces d'un objet à l'autre lorsqu'ils se percutent et c'est la méthode ***setBounce*** qui la spécifie sur les 2 axes X et Y. Plus les nombres sont proches de zéro, plus les forces seront absorbées et moins la collision aura d'effet.

```
spawnMeteor(meteorView, scene, layer) {
const meteor = this.meteorGroup.get(this.randomData.between(0, 600),
50,meteorView);
meteor.setBounce(1,1);
meteor.setCircle(32);
 ....}
```

À présent, on a un système plus réaliste. Il existe de nombreuses méthodes qui permettent de paramétrer les interactions telles que la masse, les frictions pour simuler des matériaux plus ou moins dures ou réguliers. Bref, il y a de quoi faire.

Conclusion

Phaser.js étant riche de fonctionnalités, on en a exploré une petite partie sur la base du modèle objet JavaScript ES6. Au fur et à mesure des ajouts, le code s'est alourdi avec une méthode ***update*** qui a pris du poids. Pour éviter ce type de désagrément, des ajustements seraient donc bien utiles en termes de découplage et d'architecture en retravaillant entre autres le comportement, certes sommaire, de la météorite par le biais d'une machine à états finis, ou en gérant le contrôle du joueur sous la forme d'un composant réutilisable. La méthode ***update*** doit être la plus petite possible. Mais c'est une autre histoire....

Rendu 3D sur microcontrôleur avec Gamebuino META

Jean-Marie Papillon
Ingénieur en systèmes embarqués
Président Papillon Ingénierie / Gamebuino

En 1992, Infogrames publie le jeu Alone in the Dark. C'est un jeu très immersif, tout en 3D, où l'on se déplace dans une demeure mystérieuse. Si aujourd'hui les jeux en 3D sont communs, ce n'était pas le cas à l'époque. La configuration minimale pour profiter de cette expérience était un PC équipé d'un processeur 80286 et 640 Ko de RAM, sous DOS, et une carte vidéo VGA (en fait dans le mode MCGA 320x200 pixels et 256 couleurs, comme c'était la norme à cette époque). Évidemment, aucune accélération graphique ni même de processeur de calcul à virgule flottante dans ces machines d'un autre âge. Pour arriver à ce résultat, il a fallu beaucoup de ruse à l'équipe de développement. L'essentiel des décors sont prérendu, et seuls les personnages et quelques objets sont dessinés en temps réel. Même avec ces petits raccourcis, cela reste une très belle prouesse.

https://aloneinthedark.fandom.com/wiki/Category:Alone_in_the_Dark_(1992)_images

Cet article relève le défi de créer un rendu 3D sur une Gamebuino META. Cette plateforme de jeux indépendants, bien qu'elle ne soit pas taillée pour cela, permet d'arriver à un résultat très satisfaisant, grâce à ses nombreuses ressources cachées et à l'ingéniosité de sa communauté.

La Gamebuino META est une console portable destinée à l'apprentissage de la programmation en C++ et en CircuitPython (dérivé de Python et semblable au MicroPython, NDLR). La machine est dérivée des cartes Arduino Zero avec des caractéristiques dignes d'une console portable des années 80 :

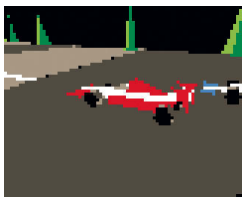
Processeur	ARM Sam21D 32 bits à 48 MHz
RAM	32 Ko
Mémoire flash programme	256 Ko
Stockage	Carte SD de 8 Go
Écran	LCD couleur 160 x 128 pixels, 65536 couleurs.
Son	Mono 44100 Hz 8 bits

La Gamebuino META est conçue, et fabriquée en France. Cocorico !

NOTE : *si vous n'avez pas la chance de posséder une Gamebuino, vous pouvez utiliser la version PC, sous QtCreator. Libre à vous de l'adapter pour un autre environnement de développement si le cœur vous en dit, sur votre propre console open source.*

Les librairies 3D

Il existe 2 librairies permettant de faire de l'affichage 3D sur Gamebuino :



G.R.O.G. (Graphics Renderer Optimized for Gamebuino) développée par Alban Rochel, aka Alban, à qui on doit l'excellente adaptation de l'univers « Retour vers le futur » avec Project88 sur Gamebuino.
GROG permet le rendu d'objets 3D solid (avec des faces pleines de couleur). La partie calcul est assez proche de l'OpenGL, ce qui la rend plus facile à prendre en main.
https://github.com/alban-rochel/grog

S.3.L. (Small 3D Library) développée par Miloslav Číž, aka drummyfish, à qui on doit l'excellent FPS multi-plate-forme Anarch.
S3L permet le rendu solid d'objets 3D, mais aussi le rendu avec des textures. La version d'origine fonctionne en résolution réduite 80x64 pixels. Nous avons adapté la librairie au rendu pleine résolution 160x128 pixels pour cet article.
https://gitlab.com/drummyfish/small3dlib/-/tree/master/

Format des objets 3D : Wavefront OBJ

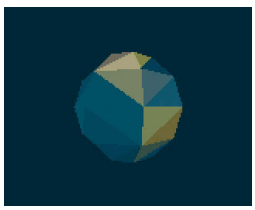
Vous connaissez forcément un bon nombre de formats utilisés pour les images en deux dimensions : BMP, GIF, PNG, JPG, TGA... la liste est longue. Pour la 3D, c'est exactement pareil ; il y a quantité de formats utilisés, dont la plupart sont propriétaires. Par exemple, le format de Blender est spécifique à Blender, mais il est open source. Le format BMP est issu de Windows, et est utilisé partout.
Ceux qui ont expérimenté l'impression 3D connaissent le format STL (pour STereo Lithogaphy). C'est un format ouvert, extrêmement populaire, mais il n'est hélas pas adapté au

rendu 3D : Le format STL est essentiellement destiné à l'impression, et n'embarque pas d'informations sur la structure logique des objets, leurs couleurs, les textures... Mais que faut-il donc pour faire un rendu 3D ?

Un rendu 3D : c'est partir d'objets définis mathématiquement dans un repère en trois dimensions, avec un X, un Y et un Z, et transformer le tout pour obtenir finalement des points en 2 dimensions que l'on peut afficher sur un écran. Même les rendus destinés à un affichage en 3D fonctionnent comme ça : on simule l'affichage en 3D en générant deux images qui seront présentées aux yeux de l'observateur. Les 2 images sont calculées à partir de deux points d'observations légèrement décalés horizontalement pour reproduire ce que voient nos yeux. C'est ensuite au cerveau de travailler pour recréer l'illusion du relief…

Un point c'est bien, mais ce n'est pas encore un objet tangible. Si on ajoute un deuxième point, on peut tracer une ligne : c'est mieux, mais encore limité. Avec un 3e point, on peut enfin créer une surface simple, le triangle. Le triangle a la particularité d'être la seule forme qui est toujours coplanaire. Cela signifie que la face définie par ce triangle est une surface parfaitement plate : ce ne serait plus forcément le cas si on utilisait 4 points, ce qui créerait des aberrations lors du rendu. C'est pour cela qu'on utilise des polynômes triangulaires en 3D.

Avec ces triangles, on peut représenter des objets en 3D dans l'espace, c'est le rendu dit « solid » ou plein. Il y a de multiples techniques pour rendre le rendu plus sexy avec ces simples triangles monochromes, comme l'ombrage de Henry Gouraud par exemple. Mais on va faire autrement en ajoutant des textures. Les textures, c'est l'artifice ultime pour représenter des objets réalistes sans utiliser des millions de triangles : plutôt que de dessiner nos triangles avec une couleur unie, on va leur affecter un petit morceau d'un dessin qui permettra de restituer des détails fins sans alourdir le modèle 3D de l'objet.



Si vous avez l'œil, vous avez remarqué que les bords de la sphère présentent des plats. On le voit mieux sur l'image de droite sans textures. Il n'y a que 80 polygones en tout, dont la moitié sont sur la face arrière invisible ici. C'est toute la beauté des textures : elles permettent d'améliorer le visuel très significativement. Ici, la résolution de la texture est de 128 x 128 pixels.

Il y a justement un format simple et adapté à cet usage : Wavefront **obj**. Il a été développé par Wavefront Technologies au milieu des années 1990.

```
# Wafefront obj format, cube example
o Object.1
v -19.999001 -20.000000 39.998001
v 19.999001 -20.000000 39.998001
v -19.999001 20.000000 39.998001
...
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 1.000000
...

f 1/1 2/2 3/3
f 8/4 3/3 2/2
f 5/5 7/6 2/7
f 2/7 1/8 5/5
...
```

Le format contient 3 types d'informations. Les lignes qui commencent par « v » pour **V**ertex, listent les points qui supportent le maillage de l'objet, avec 3 coordonnées X, Y et Z. Les lignes qui commencent par « vt » pour **V**ertex **T**extures listent des coordonnées dans la texture. Ce sont des informations en 2 dimensions qui correspondent à des coordonnées U et V, entre 0 et 1. La coordonnée vt 0 0 correspond au pixel en haut à gauche, la coordonnées vt 1 1 au point en bas à droite. Et enfin, les lignes « f » pour **F**aces décrivent les faces. Les faces font référence à des index de **V**ertex et de **V**ertex **T**extures, avec un / de séparation.

Rien de compliqué, les faces sont des triangles, et il faut 3 points XYZ pour définir la position de ses 3 angles. Pour chaque face, il faut également définir quelle est la partie de la texture qu'on souhaite y appliquer : ce sont les vt, qui associent un point dans la texture UV à chaque sommet du triangle.

Les calculs de transformation

En 3D, on utilise des matrices pour effectuer les transformations. Les matrices permettent par un procédé presque magique de faire entrer toutes les transformations nécessaires dans un seul calcul. Ces transformations permettent de transporter les objets depuis leur repère d'origine, c'est-à-dire celui qui a été utilisé lors du design, vers leur position dans la scène que l'on désire représenter. Ici, on parle de translations, mais aussi de rotations, ou de zooms. Les différents objets sont généralement représentés centrés près de l'origine [0,0,0] (dans les fichiers OBJ par exemple, mais c'est exactement la même chose quel que soit le format). Dans la scène, on doit déplacer ces objets vers des coordonnées différentes pour les placer à gauche, à droite, ou un peu partout dans l'univers 3D que l'on représente. Le déplacement, ou translation, est une fonction simple : il suffit d'ajouter un offset de déplacement sur les 3 axes X, Y et Z. Imaginons par exemple que l'on désire placer un objet en T=[10,5,0], il suffit d'ajouter 10 sur les coordonnées X de chaque point qui compose l'objet, et 5 sur chaque coordonnée Y. La 3D, c'est très simple en fait.

$$\begin{bmatrix} Xn \\ Yn \\ Zn \end{bmatrix} + \begin{bmatrix} Tx \\ Ty \\ Tz \end{bmatrix} = \begin{bmatrix} Xn + Tx \\ Yn + Ty \\ Zn + Tz \end{bmatrix}$$

La rotation est légèrement plus délicate à réaliser. Sans rotation, tous les objets seront irrémédiablement orientés dans le même sens, et le rendu ressemblerait à un univers peuplé de sprites en 2D. Bref, beaucoup de travail pour rien. On va simplement utiliser la multiplication matricielle.

$$MAT_{Rx} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix}$$

$$MAT_{Ry} = \begin{bmatrix} \cos(y) & 0 & \sin(y) \\ 0 & 1 & 0 \\ -\sin(y) & 0 & \cos(y) \end{bmatrix}$$

$$MAT_{Rz} = \begin{bmatrix} \cos(z) & -\sin(z) & 0 \\ \sin(z) & \cos(z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Pour appliquer ces matrices aux coordonnées, il suffit de faire la multiplication des coordonnées [X, Y, Z], que l'on représente comme un vecteur par la matrice désirée.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix} \begin{bmatrix} Xn \\ Yn \\ Zn \end{bmatrix} = \begin{bmatrix} 1.Xn + 0.Yn + 0.Zn \\ 0.Xn + \cos(x).Yn - \sin(x).Zn \\ 0.Xn + \sin(x).Yn + \cos(x).Zn \end{bmatrix} = \begin{bmatrix} Xn \\ \cos(x).Yn - \sin(x).Zn \\ \sin(x).Yn + \cos(x).Zn \end{bmatrix}$$

Pour multiplier un vecteur par une matrice, il faut calculer la somme des produits de chaque ligne de la matrice par la colonne du vecteur. Pour multiplier deux matrices, on fait la somme des produits de chaque ligne par chaque colonne.

En résumé, pour appliquer une rotation sur l'axe X, puis Y, puis Z, on multiplie le vecteur par la matrice X, puis par la matrice Y, puis par la matrice Z. C'est à ce moment que la magie opère : on peut limiter les calculs en calculant le produit des matrices entre elles, puis en effectuant la multiplication de la matrice résultante par le vecteur. Cela limite énormément les calculs, car il faut appliquer ces mêmes transformations à tous les points de l'objet, et il peut y en avoir des centaines ! il n'y a donc qu'une multiplication matricielle à applique à chaque point. Heureusement qu'on ne fait pas ces calculs à la main

Nous avons également besoin de déplacer les objets par translation pour les placer dans la scène en ajoutant un offset de déplacement à chaque coordonnée X, Y et Z. Second tour de magie, ce calcul peut également être intégré à la matrice !

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Xn \\ Yn \\ Zn \\ 1 \end{bmatrix} = \begin{bmatrix} 1.Xn + 0.Yn + 0.Zn + 1.Tx \\ 0.Xn + 1.Yn + 0.Zn + 1.Ty \\ 0.Xn + 0.Yn + 1.Zn + 1.Tz \\ 0.Xn + 0.Yn + 0.Zn + 1.1 \end{bmatrix} = \begin{bmatrix} Xn + Tx \\ Yn + Ty \\ Zn + Ty \\ 1 \end{bmatrix}$$

En ajoutant une quatrième dimension à nos matrices, on peut effectuer des translations par multiplication matricielle. Essayez de calculer la somme du produit des lignes de la matrice (à gauche) par la colonne du vecteur (en haut) : on obtient le vecteur au centre. Après suppression des coefficients nuls, on obtint le vecteur à droite.

$$\begin{bmatrix} Xn \\ Yn \\ Zn \end{bmatrix} + \begin{bmatrix} Tx \\ Ty \\ Tz \end{bmatrix} = \begin{bmatrix} Xn + Tx \\ Yn + Ty \\ Zn + Tz \end{bmatrix}$$

Le résultat est identique à une addition.

Notre matrice magique de dimension 4 permet ainsi en une seule opération de multiplication d'effectuer des rotations sur les 3 axes, mais aussi les translations ! C'est pour pouvoir effectuer des translations par multiplication matricielle qu'on utilise des vecteurs et matrices à 4 dimensions pour faire des calculs sur 3 dimensions. On ajoute une dimension aux vecteurs qui représentent les points, initialisée à 1, qui sert de support pour synthétiser ce calcul. Finalement, le calcul de transformation est réduit à un jeu de multiplications de matrices.

Avec un processeur disposant d'un coprocesseur mathématique, on peut évidemment effectuer ces calculs en nombres flottants. Sur un microcontrôleur, ce calcul est très coûteux en ressource, car il n'y a pas coprocesseur. C'était aussi le cas à l'époque d'Alone in the Dark. Il existe une solution : utiliser un format à virgule fixe. On utilise ainsi une partie des bits de poids faible comme une décimale. Ici, avec S3L, 9 bits servent de décimale. Ainsi, le nombre A=5.334 est représenté par A x S3L_FRACTIONS_PER_UNIT = 2731.

```
void S3L_makeRotationMatrixZXY( S3L_Unit byX, S3L_Unit byY, S3L_Unit byZ, S3L_Mat4 m)

 #define M(x,y) m[x][y]
 #define S S3L_FRACTIONS_PER_UNIT

 S3L_Unit sx = S3L_sin(byX);   // facilities
 S3L_Unit sy = S3L_sin(byY);
 S3L_Unit sz = S3L_sin(byZ);

 S3L_Unit cx = S3L_cos(byX);
 S3L_Unit cy = S3L_cos(byY);
 S3L_Unit cz = S3L_cos(byZ);

 M(0,0) = (cy * cz) / S + (sy * sx * sz) / (S * S);
 M(1,0) = (cx * sz) / S;
 M(2,0) = (cy * sx * sz) / (S * S) - (cz * sy) / S;
 M(3,0) = 0;

 M(0,1) = (cz * sy * sx) / (S * S) - (cy * sz) / S;
 M(1,1) = (cx * cz) / S;
 M(2,1) = (cy * cz * sx) / (S * S) + (sy * sz) / S;
 M(3,1) = 0;

 M(0,2) = (cx * sy) / S;
 M(1,2) = -1 * sx;
 M(2,2) = (cy * cx) / S;
 M(3,2) = 0;

 M(0,3) = 0;
 M(1,3) = 0;
 M(2,3) = 0;
 M(3,3) = S;
}
// S = S3L_FRACTIONS_PER_UNIT = 512 pour faire les calculs en virgule fixe avec 9
```

```
décimales binaires.

void S3L_makeTranslationMat( S3L_Unit offsetY, S3L_Unit offsetZ, S3L_Mat4 m)
{
 #define M(x,y) m[x][y]
 #define S S3L_FRACTIONS_PER_UNIT

 M(0,0) = S; M(1,0) = 0; M(2,0) = 0; M(3,0) = 0;
 M(0,1) = 0; M(1,1) = S; M(2,1) = 0; M(3,1) = 0;
 M(0,2) = 0; M(1,2) = 0; M(2,2) = S; M(3,2) = 0;
 M(0,3) = offsetX; M(1,3) = offsetY; M(2,3) = offsetZ; M(3,3) = S;

}
```

Et en pratique

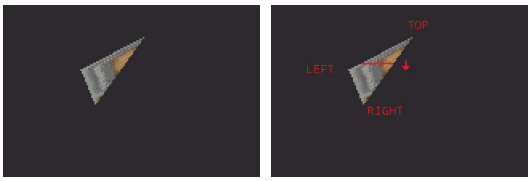
La librairie S3L prend en charge tous les calculs de matrices, mais il est important de comprendre un minimum comment tout cela fonctionne. S3L calcule travaille en fait avec 2 jeux de matrices : une première matrice est associée à chaque objet, et permet de placer ces objets dans l'univers 3D. C'est ce que l'on appelle la matrice « World » en DirectX/OpenGL. La seconde est associée à la caméra : c'est une matrice unique qui permet de faire pivoter et déplacer l'intégralité de la scène pour la placer face caméra : cette matrice est communément appelée la matrice « View ».

```
// paramétrage de la matrice camera
scene.camera.transform.translation.z = -12 * S3L_FRACTIONS_PER_UNIT;
scene.camera.transform.rotation.x = 10;

// rotation sur X de l'objet 0
scene.models[0].transform.rotation.x += 16;
```

Le rendu

Nous arrivons à la dernière étape. Nous avons des objets modélisés en 3D, que nous pouvons déplacer à notre guise dans l'environnement : il ne reste que la partie affichage.



Ce n'est pas la partie la plus complexe, mais c'est la plus gourmande en ressources. En général, on utilise une implémentation de Bresenham pour parcourir les bords du triangle. L'algorithme date des années 1960, et vous en avez certainement entendu parler : on ne va pas le détailler ici. On commence par trier les 3 points pour déterminer celui qui est le plus haut, puis on déroule en parallèle deux fois l'algorithme sur les segments de gauche et de droite pour déterminer les points gauche et droite de chaque ligne du triangle. Il suffit ensuite de remplir les points entre les deux avec une couleur unie (rendu solid), ou une couleur issue de la texture que l'on calcule par interpolation.

Z order

Il reste un dernier point important : si on affiche tous les triangles du modèle, certains vont inévitablement se superposer. Ce que nous voulons, c'est que les triangles sur le devant de la scène cachent ceux qui sont derrière. Les faces à l'arrière de l'objet devraient être cachées par les faces qui sont devant. À l'origine des temps, on utilisait un algorithme de tri : il s'agit de commencer par dessiner les faces les plus en arrière-plan, avec un Z le plus grand, puis les objets plus en avant avec des Z croissants. L'algorithme est assez simple, et très efficace si on a peu de faces à afficher. En revanche, dès que le nombre de faces des objets augmente, le nombre d'itérations pour trier ces faces à afficher augmente en proportion et le calcul devient pénalisant. De plus, ce mode d'affichage présente un petit inconvénient : il n'est pas toujours possible de déterminer si une face est devant ou derrière une autre. C'est le cas par exemple pour des faces qui se traversent mutuellement. Il n'y a pas de solution idéale. Une approche est de découper dynamiquement les faces en plusieurs sous-faces. C'est possible, mais pas facile. Heureusement, il y a une autre solution plus moderne.

Z buffer

Lorsqu'on dessine une face, on dispose des coordonnées des sommets (ou vertex). À partir de ces vertex, on parcourt les bords des faces et on remplit la surface en elle-même. Comme tout est calculé en 3D, on dispose pour chaque sommet des coordonnées X et Y, qui représentent simplement les axes horizontaux et verticaux de l'écran, mais aussi d'une coordonnée Z. C'est ce Z qui apporte une solution élégante. Si on interpole la coordonnée Z en même temps que les X et Y, on peut connaître pour chaque point de la face sa profondeur. L'astuce est d'utiliser cette coordonnée comme une couleur supplémentaire, qu'on dessine sur un plan non affiché de même dimension que l'écran : ceci construit point par point une carte de profondeur (ou height map). Ainsi, il est facile de déterminer si un point doit ou non être affiché : si la valeur de la coordonnée Z du point en cours est supérieure à celle stockée dans la carte de profondeur, le point est devant ce qui a déjà été dessiné, et on dessine ce point en mettant à jour la carte de profondeur. Dans le cas contraire, si la valeur Z est inférieure à celui stocké dans la carte de profondeur, ce point est masqué par un objet déjà dessiné, et on passe simplement au suivant.

Le point sur les ressources

Si vous travaillez sur un PC, les ressources sont plus que suffisantes pour ne pas trop se préoccuper des besoins en mémoire. Avec une petite console comme la Gamebuino META, c'est plus délicat. Pour que ce soit beau, car il est connu que les développeurs sont tous très sensibles au design, le rendu que nous effectuons ici utilise la résolution complète de l'écran, c'est-à-dire 160 x 128 pixels. Avec cette résolution et 16 bits par pixels pour coder les couleurs (5 bits de rouge, 6 bits de vert, et 5 bits de bleu), il faut 160 x 128 x 2 = 40960 octets. Il faut aussi compter avec le plan supplémentaire pour le Z buffer, qui fait également 160x128 pixels, avec des valeurs au minimum sur 8 bits pour coder 256 niveaux de profondeur. 160 x 128 x 1 = 20480 octets de plus.

Tampon image de 160 x 128 sur 16 bits	40960
Plan Zbuffer 160 x 128 sur 8 bits	20480
TOTAL	61440 octets

Finalement, il faudrait environ 60 Ko de RAM pour calculer l'image et le Z buffer avec une résolution de 160 x 128 pixels, sans compter les besoins annexes nécessaires au fonc-

Abonnez-vous à Programmez! Abonnez-vous à Programmez! Abonne

PROGRAMMEZ!

Le magazine des développeurs

NOS CLASSIQUES

1 an → 10 numéros (6 numéros + 4 hors séries)	49€*
2 ans → 20 numéros (12 numéros + 8 hors séries)	79€*
Etudiant 1 an → 10 numéros (6 numéros + 4 hors séries)	39€*
Option : accès aux archives	19€

** Tarifs France métropolitaine*

ABONNEMENT NUMÉRIQUE

PDF ..... 39€

1 an → 10 numéros
(6 numéros + 4 hors séries)

Souscription uniquement sur
www.programmez.com

Toutes nos offres sur www.programmez.com

Oui, je m'abonne

ABONNEMENT à retourner avec votre règlement à :
PROGRAMMEZ, Service Abonnements
57 Rue de Gisors, 95300 Pontoise

☐ **Abonnement 1 an : 49 €**

☐ **Abonnement 2 ans : 79 €**

☐ **Abonnement 1 an Etudiant : 39 €**
Photocopie de la carte d'étudiant à joindre

☐ **Option :** accès aux archives **19 €**

☐ Mme ☐ M. Entreprise : Fonction :

Prénom : Nom :

Adresse :

Code postal : Ville :

Adresse email indispensable pour la gestion de votre abonnement

E-mail : @

☐ Je joins mon règlement par chèque à l'ordre de Programmez !

☐ Je souhaite régler à réception de facture

*** Tarifs France métropolitaine**

Offres pouvant s'arrêter à tout moment, sans préavis **PROG Spécial 06**

Les anciens numéros de PROGRAMMEZ!

Le magazine des développeurs

236

239

240

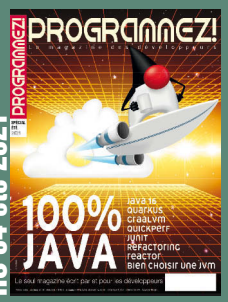
241

HS 01 été 2020

242

246

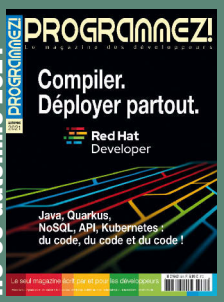
247



HS 04 été 2021

249

250



HS 05 automne 2021

Tarif unitaire 6,5 € (frais postaux inclus)

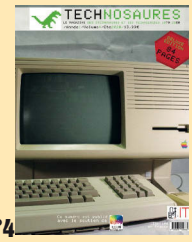
TECHNOSAURES

Le magazine à remonter le temps !

N°1

N°2

N°3



N°4

N°5

N°6

N°7 + N°8

Commandez directement sur www.technosaures.fr

☐ 235	: __ ex	☐ 241	: __ ex	☐ 247	: __ ex
☐ 236	: __ ex	☐ HS1 été 2020	: __ ex	☐ HS4 été 2021	: __ ex
☐ 239	: __ ex	☐ 242	: __ ex	☐ 249	: __ ex
☐ 240	: __ ex	☐ 246	: __ ex	☐ 250	: __ ex
				☐ HS5 automne 2021	: __ ex

Commande à envoyer à :
Programmez!
57 rue de Gisors
95300 Pontoise

soit __ exemplaires x 6,50 € = __ € ..... __ € soit au **TOTAL** = __ €

☐ M. ☐ Mme ☐ Mlle Entreprise : __ Fonction : __

Prénom : __ Nom : __

Adresse : __

Code postal : __ Ville : __

Règlement par chèque à l'ordre de Programmez ! I Disponible sur www.programmez.com

Offres pouvant s'arrêter à tout moment, sans préavis PROG spécial 06

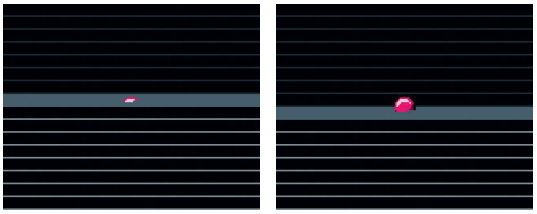
tionnement du programme, et on ne dispose que de 32 Ko de RAM. On peut utiliser des textures en 256 couleurs pour réduire les besoins en RAM : ainsi, le tampon image ne nécessite plus que 20 Ko. Le plan ZBuffer ne peut pas facilement être réduit : si on tente de coder les coordonnées de profondeur sur 4 bits, la définition en Z va devenir très limitée, ce qui fera apparaître de petits artefacts d'affichage un peu partout. On reste donc à 40 Ko environ. C'est perdu, on n'y arrivera jamais, bon week-end ! À moins que…

Microcontrôleur et cartes vidéo

Il faut s'intéresser un peu au fonctionnement de l'affichage sur les consoles basées sur une architecture Arduino. Contrairement aux consoles plus grand public, les microcontrôleurs utilisés ne possèdent pas contrôleur vidéo, c'est-à-dire un circuit électronique capable de générer un signal RGB et les signaux de balayage en continu à partir des données présentes dans une zone de mémoire. On utilise alors des écrans « intelligents » qui embarquent à la fois leur contrôleur d'affichage et la RAM d'affichage (un ST7735R pour Gamebuino META). Le microcontrôleur calcule ainsi l'image dans sa RAM, puis transfère les données RGB vers la RAM du contrôleur d'écran. Une fois le transfert terminé, on peut effacer le tampon RAM du microcontrôleur pour calculer l'image suivante : l'écran continue d'afficher la même image. Cette technique est appelée double buffering : pendant qu'une image est affichée à l'écran, on calcule la suivante. Le double buffering est une technique plus que courante dans le développement de jeux vidéo : il permet de masquer les opérations de construction de l'image : on ne visualise que les images finales, sans voir apparaître l'affichage successif des diverses parties qui composent l'image, notamment lorsque plusieurs sprites se superposent. Cela nécessite simplement de disposer de deux fois la RAM composant une image.

Le slicing

Andy O'Neill a proposé une solution originale pour profiter de la résolution complète de l'écran de la console bien que la RAM disponible ne le permette pas a priori. Andy a également développé l'émulateur en ligne vous pouvez tester sur le site pour exécuter les jeux développés pour Gamebuino META, sans disposer de console. Sa solution est aussi élégante que performante.



(Merci à Stéphane Calderoni pour l'illustration).
L'article complet est disponible ici :
https://m1cr0lab-gamebuino.github.io/gb-shading-effect/fr/
Au lieu d'utiliser un tampon d'une image complète dans la RAM du microcontrôleur, et un second tampon dans la RAM de l'écran, on utilise deux tampons qui contiennent chacun une petite bande de l'image complète dans la RAM du microcontrôleur, par exemple de 16 lignes. Pendant qu'on dessine sur un des tampons, le second est transféré par un contrôleur DMA vers l'écran. (Un contrôleur DMA, pour Direct Memory Access est un module qui permet de transférer des blocs de données entre la RAM et un périphérique sans faire intervenir le processeur). On répète l'opération autant de fois que nécessaire pour actualiser entièrement la mémoire de l'écran, en échangeant à chaque itération le rôle des deux tampons coté MCU. Avec des tampons de 16 lignes, on actualise les 128 lignes d'affichage de l'écran en 8 fois 16 lignes.
On va utiliser ici une petite variante avec deux tampons différents dans la mémoire du MCU : le premier sera le tapon utilisé pour faire un rendu en 256 couleurs. Le second sera utilisé pour le transfert en RBG565 vers la RAM de l'écran. Entre chaque itération, il suffit de convertir le premier tampon en couleurs indexées vers le second en couleurs RGB en utilisant une palette. L'empreinte mémoire devient alors :

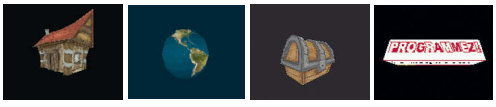
Tampon « dessin » de 160 x 32 en 8 bits	5120 octets
Tampon de transfert 160 x 32 en 16 bits RGB	10240 octets
Plan Zbuffer de 160 x 32 en 8 bits	5120 octets
TOTAL	20480 octets

Victoire : il reste environ 12 Ko pour les autres besoins. Cela tombe bien, car il reste un dernier poste consommateur de mémoire. Nous avons étudié comment transformer les coordonnées des objets pour les représenter dans la position désirée : cela nécessite de stocker les coordonnées d'arrivée de chaque vertex des objets. Avec seulement 128 vertex, dont chacun nécessite 4 dimensions en 32 bits, il faut encore 2 Ko de RAM. Il reste 10 Ko pour le reste du programme.

```
Void draw()
{
 S3L_newFrame() ;          // initialisation de la lib pour une nouvelle image
 S3L_transformScene(scene) ;// calcul des transformations
 for ( uint8_t u8_page_index = 0 ; u8_page_index < 128/32 ; u8_page_index++ )
 {          // l'affichage d'une image complète nécessite 4 itérations de 32 lignes
  S3L_zBufferClear();
  pu8_screen = gb_get_video_buffer(u8_page_index);
  memset( pu8_screen, 0, 160*128 ) ;          // effacement du tampon-écran
  S3L_drawTransformedScene( scene, u8_page_index ) ;    // rendu de la scèné
  if ( !u8_page_index )
  {
   gb_end_update() ;     // on restitue le contrôle de l'écran au système de la console
   while ( !gb.update() ) ;// gb.update() effectue les petites taches de fond de la console
  }
  gb_update(u8_page_index) ;  // conversion du tampon 256 couleurs en RGB
et transfert vers l'écran.
}
```

Extrait du code : boucle d'affichage principale

Conclusion



Avec ces petites optimisations, on arrive à des performances honorables.

Maison	Vertex	Faces	Framerate
Coffre	127	200	12.77 fps
Planète Terre	42	80	17.41 fps
Logo Programmez	8	12	18.54 fps (au zoom maxi)

Si vous voulez adopter une Gamebuino et soutenir le projet, profitez de 10 % de réduction et des frais de port gratuits avec le code « PROGRAMMEZ » sur www.gamebuino.com

Un jeu pour le test logiciel ?

Les jeux sont présents presque partout, à tout moment de notre vie mais aussi dans le monde animal où ce dernier sert à mesurer et à établir un ordre établi. Cette présence montre un intérêt au-delà de l'amusement lié à cette activité. S'il est un endroit où les jeux ne sont pas (encore) vraiment présents c'est le milieu professionnel et notamment celui du test logiciel ou plus largement de l'IT.

Marc Hage Chahine

Expert méthodes et outils chez Sogeti. Créateur du blog La taverne du testeur.

Mais, avant toute chose, il me paraît important de se mettre d'accord sur ce qu'est exactement un jeu car, comme vous l'aurez compris, les "serious games" ne sont pas vraiment des jeux !

Qu'est-ce qu'un jeu ?

Il y a de nombreuses définitions au terme « jeu », néanmoins si je devais retenir une définition permettant de déterminer si l'on parle d'un jeu ou non ce serait une combinaison de 4 éléments.

Le jeu en lui-même est universel dans l'idée où il existe toujours quelque part un jeu adapté à sa condition, ses compétences et ses connaissances.

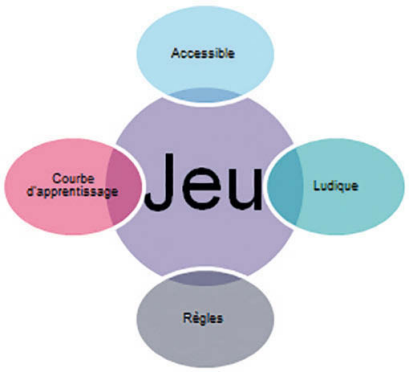
Pour moi, un jeu doit :

- Être **Accessible**: à part exception ou contraintes spécifiques, un jeu doit être compréhensible et jouable par un grand nombre de personnes
- Être **Ludique** : un jeu doit être amusant, donner l'envie d'y (re)jouer (par exemple avec un challenge et l'envie de se dépasser)
- Avoir un **But** et des **règles** : il doit y avoir un objectif et un cadre pour y arriver. Il est bon de noter que certaines règles, notamment dans les jeux vidéo et jeux de société "legacy", peuvent se découvrir progressivement
- Proposer une **courbe d'apprentissage** : on doit pouvoir s'améliorer avec l'expérience

Avec cette définition, on remarque que le jeu englobe différentes variantes comme les jeux vidéo, les jeux de société, les jeux de cartes mais aussi certains sports (on « joue » au foot) :

On remarque également que les ateliers organisés dans le cadre professionnel ne sont pas des jeux, comme les « serious games » notamment à cause du manque de « fun » mais aussi de par l'absence de courbe d'apprentissage (peu de chose à apprendre après 1 partie) et une nécessité fréquente d'avoir des connaissances en amont sur un métier en particulier.

Au final je résume un jeu avec cette image :



Savoir ce qu'est un jeu c'est bien mais quels sont les apports des jeux ? Pourquoi ces « jeux » qui peuvent sembler être une perte de temps pour certains (car les jeux ne "produisent" rien) ont-ils tant de succès ?

Les apports des jeux

Comme mentionné précédemment, les jeux sont très présents. Ils sont présents à tout âge de notre vie (avec des jeux

Jeu	Ludique	Règles[OD1]	Apprentissage
Football	Entraide (équipe) Accomplissement Courir…	+ de buts que son adversaire, 11 joueurs, 2 équipes, limites terrain…	Stratégie, communication, tirs, passes…
Echecs	Challenge intellectuel	Déplacements des pièces, 2 joueurs, chacun son tour	Stratégies, anticipation, calculs
Mario	Aller aussi loin que possible, speed runs…	Sauver Peach Encadrées et définies par le jeu!	Maîtrise de mario, différents ennemis et mondes
Bataille	Aspect aléatoire, imprévu	La carte la plus forte l'emporte, si égalité: bataille	Reconnaître les valeurs les plus élevées
Pétanque	Résultat des actions, décisions…	Calcul des points, boules les plus proches, tour de jouer…	Précision, stratégie (tirer ou pointer)…
Jungle speed	Situations	Attraper le totem le plus vite possible au bon moment	Reconnaître les situations, anticiper, attraper le totem…

adaptés à différents âges) depuis des millénaires avec des pierres peintes sculptées datant de 5000 ans. Cette attirance pour le jeu est forcément liée à des avantages procurés par les jeux… qui s'avèrent donc beaucoup plus utiles que ce qui est véhiculé par certaines idées reçues comme le fait que les jeux sont uniquement un passe-temps pour les enfants et qu'un adulte ne devrait pas jouer.

Les apports des jeux sont nombreux, je vois néanmoins 4 axes qui me paraissent particulièrement importants et fréquents dans les jeux. En effet, les jeux nous apportent régulièrement un ou plusieurs des éléments ci-dessous :

- Des moments de **partage** : on partage du temps avec d'autres personnes (des amis, sa famille, ses enfants, des inconnus avec la même passion…)
- De la **joie** : cela permet de penser à autre chose et de profiter de moments agréables
- La discipline pour **suivre des règles** : on apprend à évoluer dans un cadre, repérer des limites (ou les contourner pour les tricheurs), savoir ce qui est autorisé… tout en ayant pour but d'atteindre l'objectif du jeu. Sur cet aspect, cela correspond très bien à la vie de tous les jours... et à l'éducation de ses enfants
- Un **développement de compétences** : pour s'améliorer à un jeu il faut s'adapter et apprendre de ses erreurs. Ces adaptations développent des compétences spécifiques à chaque jeu.

Prenons l'exemple d'un jeu classique et très accessible comme les mille bornes. Ce jeu offre bien :

- Des moments de **partage** grâce à du temps passé avec ses amis, enfants ou parents à tout âge.
- De la **joie** à travers les « vacheries », les « coup fourrés », les surprises du hasard ou encore la victoire (les parties courtes multiplient les chances de gagner sur une session de jeu)
- Un **suivi des règles** assez strictes et dépendant beaucoup du contexte (feu vert, limitations, vacheries…)
- Le **développement de compétences** comme le **calcul** (arriver jusqu'à 1000), la **stratégie** (garder ses « bottes » pour avoir un effet de surprise, garder en main les cartes pouvant aider notre adversaire…) ou encore **l'anticipation** (garder des cartes utiles pour contrer des vacheries adverses…)

Lorsque l'on voit tous ces apports et la diversité des jeux, il semble potentiellement intéressant de se pencher sur la possibilité de faire un jeu dédié au test logiciel (ou tout autre aspect du l'industrie du logiciel). Il faut alors se demander ce que l'on voudrait réussir à faire avec un ou des jeux pour le test.

Pourquoi un jeu pour le test ?

Les raisons peuvent être multiples. J'en vois principalement 3 : La première serait pour introduire le test de manière ludique. Le test est encore assez peu connu (je n'ai pas encore rencontré d'enfants (et même de lycéens) souhaitant devenir testeur. Un jeu permettant d'introduire ce métier permettrait de :

- Faire connaître ce métier et de susciter des vocations
- Aider des professionnels à bien débuter dans le test
- Faire comprendre à des non professionnels certains aspects de ce métier

Une deuxième serait plus opérationnelle en proposant de développer des compétences spécifiques. Attention, il ne faut pas tomber dans les Serious Games mais bien identifier des compétences spécifiques (comme la capacité à bien faire passer un message) et les mettre dans un jeu qui n'est pas forcément sur un thème lié à l'IT. Les compétences nécessaires au test sont multiples, on peut par exemple penser à des :

- Compétences de planification
- Compétences de communication et de collaboration
- Compétences de recherche et d'observation
- Compétences de synthétisation….

La troisième raison que je propose est de donner une bonne image du test en proposant des moments de partage et des souvenirs positifs liés au test.

Ceci étant posé, il reste maintenant le plus difficile, comment faire un jeu pour le test ?

Comment faire un jeu pour le test ?

Je tiens ici à préciser qu'un jeu pour le test ne doit pas forcément être dans un environnement de test ou de développement logiciel. Ceci est d'autant plus vrai pour les jeux conçus pour développer des compétences. De ce point de vue il existe déjà des jeux permettant d'améliorer des compétences nécessaires dans le monde du test. On peut par exemple penser à des jeux comme le Times'Up qui fait travailler sa communication. En effet, le Times'Up car pousse à bien choisir ses définitions et descriptions pour faire deviner des mots ou des personnes ou notions… avant de pousser à une forte sélection de ses mots et de finir avec un langage corporel sans mot.

De plus, il me semble très important de rappeler que concevoir des jeux, c'est un métier à part entière et qu'il peut être nécessaire, si l'on veut développer un bon jeu, de travailler avec des professionnels ou des passionnés.

Dans tous les cas, pour réussir à faire un bon jeu pour le test, il faut passer par des étapes. Ces étapes sont, au minimum :

- **La définition du public cible** : on ne fera pas le même jeu pour des enfants, pour des adolescents ou pour des adultes ! Les mécaniques devront s'adapter au public
- **L'objectif du jeu** : un seul jeu ne sera pas suffisant pour tout faire comprendre et tout apprendre du métier de testeur. Comme dans le monde du test, il faudra faire des choix ! Des choix comme les compétences que l'on souhaite développer, le niveau de compétence que l'on souhaite atteindre mais surtout ce que l'on veut mettre en avant !
- **La conception du jeu** : c'est la partie la plus technique ! Cette conception devra s'orienter pour répondre au contexte (à travers des mécaniques) tout en s'assurant de garder l'aspect ludique.
- **Les tests du jeu** : c'est un peu notre bêta test. Cela permet de voir la réception du jeu mais aussi de faire les ajustements nécessaires.

J'ai tenté de faire l'exercice sans aller jusqu'à la fin de la conception ou encore les tests. Ces tentatives nous donnent cependant une idée de ce à quoi pourrait ressembler un jeu pour les tests :

Un jeu de rôle coopératif

Exemple de jeux de rôle coopératif: Andor, Aeon's End, Zombicide... **Figure 1**

Un jeu de gestion de ressources

Exemple de jeux de gestion de ressources: Architectes, Root, Agricola, Terraforming Mars. **Figure 2**

Conclusion

Les jeux ne sont pas un passe-temps pour les enfants. Les jeux sont très importants dans notre développement, c'est pour cela qu'ils sont universels et intemporels. Les jeux aident à développer des compétences sans avoir l'impression de « travailler ».

Utiliser la puissance des jeux pour le test logiciel (ou tout autre domaine professionnel) dispose d'un très fort potentiel qu'il serait dommage de négliger.

Il ne faut cependant pas perdre de vue que faire un « bon » jeu n'est pas simple. Il faut que le jeu soit ludique mais aussi qu'il permette de développer des compétences spécifiques.

Un jeu pour le test logiciel reste totalement réalisable et j'ai déjà vu des initiatives allant dans ce sens pour de la formation ou de la vulgarisation. Dans ces expériences les jeux inspirés des 1000 bornes et d'un autre jeu de cartes étaient encore en phase de test mais les bases (objectifs du jeu pour les testeurs) étaient posées.

Il ne faut pas oublier que lorsque l'on fait un jeu pour le domaine professionnel (dans notre cas le test logiciel) il faut voir plus loin que le « serious game » mais bien proposer un jeu auquel même les non testeurs voudront jouer et rejouer.

La meilleure manière d'y parvenir c'est de se rapprocher d'initiative déjà existante ou d'en initier. De proposer des jeux, des concepts et de les essayer. Au final il y aura sûrement de nombreux jeux de créés à partager. Ces jeux seront complémentaires car les compétences à développer ou la manière de les développer seront différentes.

A vos jeux, prêts, testez!

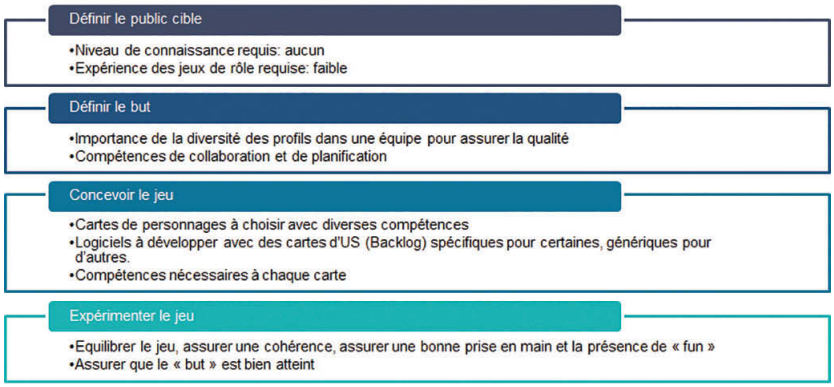


Figure 1

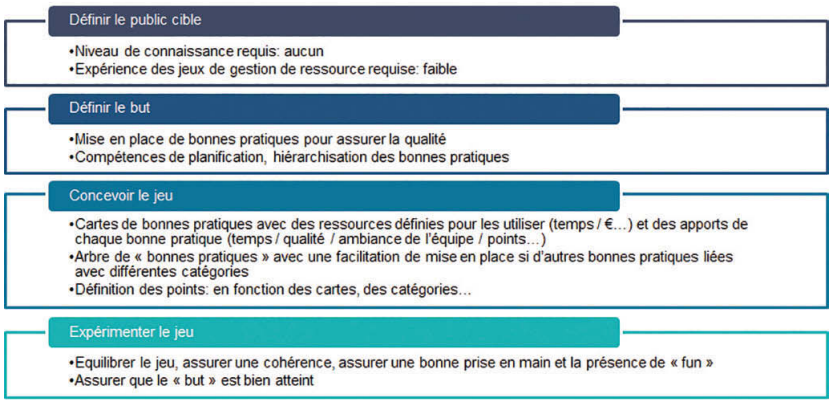


Figure 2

Grégory Bersegeay
MVP Embarcadero (Delphi), développeur fullstack, conférencier.

Shoot'm Up multi-plate-forme avec Delphi

La mode étant au rétro gaming, je vous propose de réaliser un jeu de type « shoot'm up » multi-plate-forme en 2D comme au bon vieux temps. Nous allons tout développer de A à Z sans utiliser de moteur de jeu, car, en tant que développeur, il est toujours intéressant de comprendre les bases.

Pour réaliser le jeu, nous utiliserons Delphi sans composants additionnels. Ce projet est donc compatible avec l'édition Community de Delphi gratuite (sous certaines conditions et pour une utilisation personnelle). Les sources du projet sont disponibles au lien indiqué en fin de l'article. Delphi permet de développer toutes sortes d'application facilement, rapidement et sans avoir besoin d'une multitude de couches diverses et variées comme cela peut être le cas avec d'autres outils de développement.

N'étant pas graphiste et afin d'augmenter le côté nostalgique, j'ai repris les sprites du vaisseau et du module complémentaire de Thunder Force 4 sur Megadrive (un célèbre représentant des shoot'm up). Et comme je ne suis pas non plus musicien, j'utilise des sons provenant des kits de Kenney (https://www.kenney.nl/) et la musique est de Oblidivm (https://opengameart.org/content/space-shooter-music).

Gameplay

Notre jeu sera donc un jeu de tir à scrolling horizontal. Le joueur pilotera un vaisseau et devra détruire et/ou éviter des vagues de vaisseaux aliens. Scénario pas très original, mais ce genre de jeu a connu son heure de gloire dans les années 80/90.

Sur Desktop, le joueur jouera au clavier à l'aide des flèches pour déplacer le vaisseau et de la touche CTRL pour tirer. Sur les plateformes mobiles (iOS et Android), des boutons seront affichés afin de pouvoir jouer.

Le joueur perd dès qu'il heurte un vaisseau ennemi. Il y a trois types de vaisseaux ennemis qui auront tous le même design, mais reconnaissables à leur couleur indiquant leur résistance. En effet, les vaisseaux ennemis bleus ne disposeront que d'un seul point de vie, les verts en auront 4 et les violets 8. Chaque ennemi détruit rapportera des points en fonction de son type : les bleus rapporteront 100 points, les verts 200 et les violets 500.

Enfin, deux types bonus seront gérés. Le joueur devra faire passer le vaisseau sur le bonus pour l'attraper. Un bonus sera affiché à l'écran toutes les 20 secondes. Il y aura deux types de bonus : ceux rapportant des points, ceux permettant d'obtenir un tir plus puissant.

Voilà pour les éléments de gameplay : c'est simple, mais ça doit tenir dans cet article.

Création du projet sous Delphi

Comme pour mes articles précédents, nous allons créer un projet de type « Application multipériphérique » afin d'utiliser le framework FMX. Il sera ainsi possible de compiler le jeu pour Windows, macOS, Linux, iOS et Android.

Nous créons l'ossature de l'interface graphique de notre jeu à l'aide du concepteur graphique : il suffit de déposer des composants, les placer et d'ajuster certaines de leurs propriétés via l'inspecteur d'objet. (**figure 1**).

Un petit zoom sur la fenêtre « Structure » montre la hiérarchie de ces composants graphiques (**figure 2**).

Figure 1

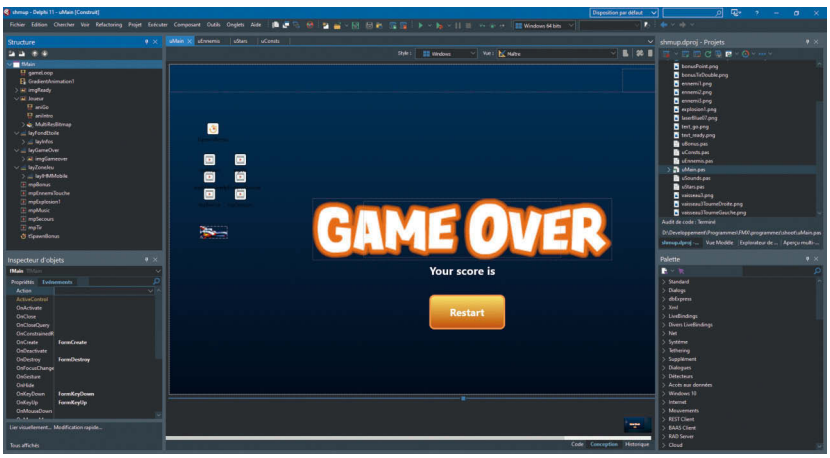
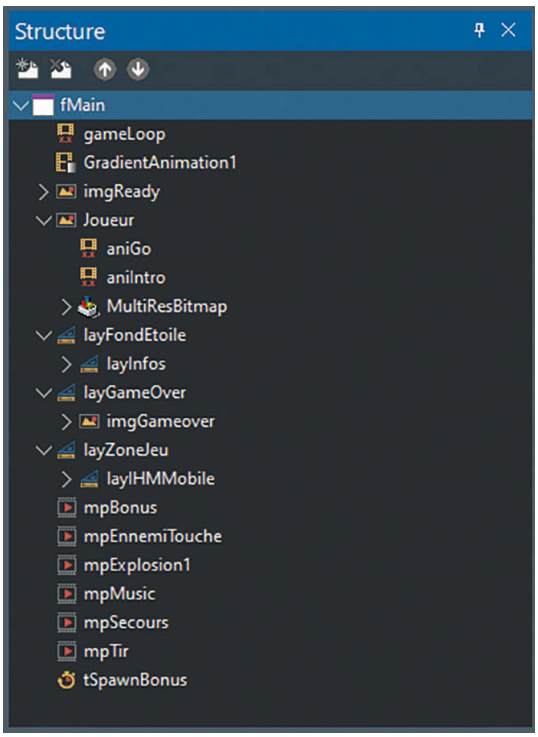


Figure 2



Pour notre jeu, nous n'utiliserons qu'une poignée de composants. En effet, nous utiliserons essentiellement des **TAnimation**, des **TLayout**, des **TImage** et des **TMediaPlayer**. En effet, ces composants fournis en standard avec Delphi nous permettront de :

- gérer toutes sortes d'animation (ces dernières sont threadées);
- organiser les plans d'affichage ;
- afficher des images ;
- jouer des sons et des musiques.

Le composant nommé **gameLoop** est un **TFloatAnimation** qui servira de boucle principale. Dans son événement **OnProcess**, nous gérerons les interactions du joueur, les ennemis, les tirs, etc.
Le composant **GradientAnimation1** sera associé à la propriété **Fill.Gradient** de la fiche principale **fMain**. Le fond de la fenêtre ainsi paramétré affichera un dégradé de couleurs animé.
L'image **imgReady** affichera les images « Ready » ou « Go » en fonction du besoin. L'image nommée **Joueur** correspondra au sprite du vaisseau piloté par le joueur. Deux composants de type **TFloatAnimation** lui sont affectés : **aniIntro et aniGo**. Il s'agit de deux petites animations qui s'enchaînent et qui permettent de déplacer le vaisseau et de gérer l'affichage de l'image « Ready » ou « Go » en début de partie.
Nous avons ensuite trois composants de type **TLayout** :

- **layFondEtoile** : placé en arrière-plan, il servira à dessiner des étoiles ;
- **layZoneJeu** : placé devant le **layFondEtoile**, il contiendra les éléments de jeu (ennemis, bonus et tirs).
- **layGameOver** : il s'agit de l'écran de « Game over ». Il s'affichera à l'écran lorsque le vaisseau du joueur sera touché.

D'autres **TLayout** seront utilisés pour afficher le score ou l'interface d'interaction visible uniquement sur les plateformes mobiles.
Nous utiliserons également plusieurs **TMediaPlayer** afin de pouvoir jouer la musique et des sons en même temps.
Enfin, le **TTimer** nommé **tSpawnBonus** servira à afficher un bonus toutes les 20 secondes.
Notre projet contiendra 6 unités :

- uMain.pas : la fiche principale qui contiendra l'écran de jeu, les composants graphiques et le code principal du jeu ;
- uBonus.pas : unité permettant de créer et déplacer un bonus ;
- uConsts.pas : unité contenant les constantes utilisées dans le jeu ;
- uEnnemis.pas : unité permettant de gérer les ennemis ;
- uSounds.pas : le nécessaire charger et jouer un son ;
- uStars.pas : unité permettant de gérer le défilé d'étoiles.

Les présentations étant faites, entrons à présent dans le détail !

Gestion de l'arrière-plan

Nous avons vu que l'arrière-plan sera géré en deux temps : le dégradé d'arrière-plan animé et un défilé d'étoiles.
Pour le dégradé animé, aucun code n'est nécessaire (merci Delphi). En effet, nous utilisons la propriété **Fill** de la fiche à laquelle on associe le **TGradientAnimation** et le tour est joué.
Pour le défilé d'étoiles, en revanche, il va falloir coder un peu… Comme évoqué précédemment, le code gérant le fond étoilé se trouve dans l'unité uStars. Nous y retrouvons la classe **TStar** qui permet de définir une étoile à partir de sa position, sa vitesse de déplacement et d'une méthode **Draw** permettant de dessiner l'étoile :

```
TStar = class
 Position: TPointF;
 Speed: single;
 procedure Draw(const c: TCanvas);
end;
```

La méthode **Draw** prend en paramètre le canevas où sera dessinée l'étoile et ensuite, en fonction de la vitesse que l'on va donner à l'étoile, nous allons lui attribuer une couleur. Ainsi, les étoiles les plus lentes seront en gris foncé, les moyennement rapides seront en gris clair et les plus rapides seront blanches. Cela donnera un petit effet de profondeur à notre défilé d'étoile.
Une fois la couleur déterminée, nous dessinons l'étoile qui n'est rien de plus qu'une petite croix de 12 pixels autour de sa position. Voici le code de méthode **Draw** :

```
procedure TStar.Draw(const c: TCanvas);
begin
 if Speed > 6 then
  c.Stroke.Color := TAlphaColorRec.white
 else if Speed > 5 then
  c.Stroke.Color := TAlphaColorRec.MedGray
 else
    c.Stroke.Color := TAlphaColorRec.Slategray;
 c.Stroke.Thickness := 1;
 c.Stroke.Kind := TBrushKind.Solid;
 c.DrawLine(PointF(Position.X-6, Position.Y), PointF(Position.X+6, Position.Y), 1);
 c.DrawLine(PointF(Position.X, Position.Y-6), PointF(Position.X, Position.Y+6), 1);
end;
```

La seconde classe que l'on trouve dans l'unité **uStars** est la classe **TStars** :

```
TStars = class
 private
  FWidth, FHeight: single;
  FStars: TList<TStar>;
  FNbStars : integer;
  procedure setNbStars(const Value: integer);
  procedure CreateStars;
 public
  constructor Create(const AWidth, AHeight: single; nbStars : integer);
  destructor Destroy; override;
  procedure Update;
  procedure Draw(const c: TCanvas);
  property StarsCount : integer read FNbStars write setNbStars;
  property Width : single read FWidth write FWidth;
  property Height : single read FHeight write FHeight;
end;
```

Cette classe contient une liste de **TStar**. Nous allons ainsi créer un ensemble de **TStar** dans une zone définie par une

largeur (propriété **width**) et une hauteur (propriété **height**). L'ensemble contiendra **StarsCount** étoiles.
La méthode **Draw** consiste à invoquer la méthode **Draw** de chaque étoile de l'ensemble :

```
procedure TStars.Draw(const c: TCanvas);
begin
 for var s in FStars do
  s.Draw(c);
end;
```

La méthode **update** permet quant à elle de mettre à jour la position de chaque étoile de l'ensemble en fonction de sa vitesse. Si la position de l'étoile sur l'axe X est inférieure à 0 (l'étoile n'est plus visible, car elle est sortie de l'écran sur la gauche), alors nous la replaçons à droite de l'écran à une hauteur (axe Y) aléatoire :

```
procedure TStars.Update;
begin
 for var s in FStars do begin
  if s.Position.X < 0 then s.Position := PointF(FWidth, random(round(FHeight)))
  else s.Position.X := s.Position.X - s.Speed;
 end;
end;
```

Enfin, la méthode **createStars** permet de créer l'ensemble des étoiles en les positionnant aléatoirement sur la zone définie par les propriétés **Width** et **Height** et en donnant une vitesse aléatoire.

```
procedure TStars.CreateStars;
begin
 for var i := 0 to FNbStars-1 do begin
  var s := TStar.Create;
  s.Position := PointF(random(round(FWidth)), random(round(FHeight)));
  s.Speed := 5 + random(3);
  FStars.Add(s);
 end;
end;
```

Notre défilé d'étoiles est prêt, il reste à créer l'objet et à le dessiner à une fréquence régulière pour qu'il soit effectif.
Pour ce faire, dans l'événement **OnCreate** de la fiche principale, nous créons un objets nommé stars :

```
stars := TStars.Create(jeuWidth, jeuHeight, NB_ETOILES);
```

NB_ETOILES étant une constante définie dans l'unité **uConsts** qui correspond au nombre d'étoiles que l'on souhaite gérer. Par défaut, j'ai initialisé cette valeur à 60.
Pour la mise à jour des étoiles, nous le ferons simplement dans l'événement **OnPaint** du **layFondEtoile** en calculant les futures positions des étoiles puis en les dessinant :

```
procedure TfMain.layFondEtoilePaint(Sender: TObject; Canvas: TCanvas; const
                                    );
begin
 stars.Update;
 stars.Draw(layFondEtoile.Canvas);
end;
```

Nous avons notre fond étoilé généré aléatoirement avec trois niveaux de profondeur simulés qui défile perpétuellement de droite à gauche.

Le plan de jeu

Le plan de jeu représenté par le **layZoneJeu** va être le layout où seront dessinés les ennemis, les tirs et les bonus.
Nous allons calquer la gestion des ennemis sur la gestion des étoiles. Ainsi le code de gestion des ennemis est regroupé dans l'unité uEnnemis qui regroupe deux classes : **TEnnemi** et **TEnnemis**.
Un ennemi sera représenté par un **TEnnemi** dont voici le détail :

```
TEnnemi = class(TImage)
 private
  fSpeedX : single;
  fLife, fPoints : integer;
  fAnimationImage : TBitmapListAnimation;
  fAnimationY : TFloatAnimation;
 public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property VitesseX : single read fSpeedX write fSpeedX;
  property PointVie : integer read fLife write fLife;
  property Points : integer read fPoints write fPoints;
  property Animation : TBitmapListAnimation read fAnimationImage write
fAnimationImage;
  property AnimationY : TFloatAnimation read fAnimationY write fAnimationY;
end;
```

Un ennemi sera donc dérivé du **TImage** standard auquel nous ajoutons les propriétés :

- **VitesseX** : sa vitesse de déplacement sur l'axe des abscisses ;
- **PointVie** : le nombre de points de vie de l'ennemi (c'est-à-dire sa résistance) ;
- **Points** : le nombre de points que rapportera l'ennemi une fois détruit par le joueur ;
- **Animation** : un **TBitmapListAnimation** qui permet d'utiliser une image de type « sprite cheet » en guise d'image. Une image de type « sprite cheet » est une image contenant plusieurs images plus petites. Chaque image compose alors une étape de l'animation (la figure 3 représente la sprite cheet utilisée pour les ennemis de notre jeu). L'image affichée résultante sera alors uniquement l'image de l'étape en cours de l'animation. L'association d'un **TBitmapListAnimation** avec un **TImage** permet très facilement de gérer des sprites animés avec Delphi.
- **AnimationY** : est un **TFloatAnimation** qui permettra de gérer le mouvement de l'ennemi sur l'axe des ordonnées (axe Y).

Figure 3

Le constructeur de la classe **TEnnemi** créer un ennemi avec un peu d'aléatoire :

```
constructor TEnnemi.Create(AOwner: TComponent);
begin
 inherited;
 height := 48;  // x3
 width := 57;
 fAnimationImage := TBitmapListAnimation.Create(nil);
 fAnimationImage.Parent := self;
 fAnimationImage.PropertyName := 'bitmap';
 fAnima fAnimationY := TFloatAnimation.Create(nil);
 fAnimationY.Parent := self;
 fAnimationY.Duration := 1+random(4);
 fAnimationY.PropertyName := 'position.Y';
 fAnimationY.Loop := true;
 fAnimationY.AutoReverse := true;
 fAnimationY.AnimationType := TAnimationType.InOut;
 fAnimationY.Interpolation := TInterpolationType.Sinusoidal;
 fAnimationY.StartValue := Position.Y;
 fAnimationY.StopValue := Position.Y + 100 + random(200);
 fAnimationY.Delay := random;
```

```
 VitesseX := 3 + random(5);
 PointVie := 1;
 fPoints := 100;
end;
```

En effet, nous créons l'animation sur l'axe Y (**fAnimationY**) avec une durée, un délai et une amplitude aléatoires. De même, la vitesse de l'ennemi sur l'axe des abscisses est aléatoire.

À noter, j'ai arbitrairement multiplié par 3 la taille du sprite d'origine afin qu'il soit bien visible sur un écran moderne doté d'une définition élevée. En effet, comme précisé en début d'article, le sprite d'origine provient d'un jeu Megadrive et cette dernière avait une résolution de 320x224 et gérait des sprites de maximum 32x32 pixels.

Comme pour les étoiles, la classe **TEnnemis** va permettre de gérer un ensemble d'ennemis dans la zone de jeu. Voici la définition de cette classe :

```
TEnnemis = class
 private
  FWidth, FHeight: single;
  FNbEnnemis : integer;
  fParent : TFMXObject;
 public
  constructor Create(const AWidth, AHeight: single; nbEnnemis : integer; parent : TF
MXObject);
  destructor Destroy; override;
  var listeEnnemis: TList<TEnnemi>;
  procedure CreateEnnemis;
  procedure Update;
  property Width : single read FWidth write FWidth;
  property Height : single read FHeight write FHeight;
end;
```

Nous retrouvons des propriétés **Width** et **Height** qui détermineront la zone d'action des ennemis. Ces derniers arriveront par vague de la droite de l'écran et se dirigeront vers la gauche. L'ennemi disparaîtra lorsqu'il sera détruit par le joueur, ou s'il parvient à sortir de l'écran par la gauche.

Lorsque tous les ennemis d'une vague auront disparu alors une nouvelle vague d'ennemis sera générée en incrémentant le nombre d'ennemis constituant la vague.

En complément des propriétés **Width** et **Height**, la classe **TEnnemis** dispose des méthodes **CreateEnnemis** et **Update**. Là encore de la même manière que pour les étoiles, la méthode **CreateEnnemis** permet de créer une liste de **TEnnemi** contenant **FNbEnnemis**.

```
procedure TEnnemis.CreateEnnemis;
begin
 for var i := 1 to FNbEnnemis do begin
  var unEnnemi := TEnnemi.Create(FParent);
  unEnnemi.parent := FParent;
  unEnnemi.Position.X := FWidth + 50 + random(round(FWidth));
  unEnnemi.Position.Y := 40 + random(round(FHeight)-300);

  case random(9) of
   1,6: begin
     unEnnemi.fAnimationImage.AnimationBitmap := fMain.ennemi2;
     unEnnemi.PointVie := 4;
     unEnnemi.fPoints := 200;
     unEnnemi.fAnimationImage.Duration := 0.3;
    end;
   3: begin
     unEnnemi.fAnimationImage.AnimationBitmap := fMain.ennemi1;
     unEnnemi.PointVie := 8;
     unEnnemi.fPoints := 500;
     unEnnemi.fAnimationImage.Duration := 0.2;
    end;
   else begin
     unEnnemi.fAnimationImage.AnimationBitmap := fMain.ennemi3;
     unEnnemi.PointVie := 1;
     unEnnemi.fPoints := 100;
     unEnnemi.fAnimationImage.Duration := 0.4;
    end;
  end;

  unEnnemi.fAnimationImage.AnimationCount := 8;
  unEnnemi.fAnimationImage.Loop := true;
  unEnnemi.fAnimationImage.Enabled := true;
  unEnnemi.fAnimationY.StartValue := unEnnemi.Position.Y;
  unEnnemi.fAnimationY.StopValue := unEnnemi.Position.Y + 100 + random(200);
  unEnnemi.fAnimationY.Start;

  listeEnnemis.Add(unEnnemi);
 end;
end;
```

Nous créons **FNbEnnemis** ennemis que nous plaçons au-delà de la droite de l'écran et une nouvelle fois avec un peu d'aléatoire, nous créons des ennemis bleus (les plus fréquents, mais les moins résistants et rapportant peu de points), des verts moins nombreux et des violets encore moins nombreux.

La méthode **update** permet de gérer l'affichage des ennemis : on détruit un ennemi ayant dépassé la gauche de l'écran et s'il n'y a plus d'ennemis dans la liste des ennemis, alors on crée une nouvelle vague d'ennemis (en augmentant

le nombre d'ennemis de la vague) :

```
procédure TEnnemis.Update;
begin
 for var e in listeEnnemis do begin
  e.Position.X := e.Position.X - e.VitesseX;
  if (e.Position.X < -50) or (e.PointVie = 0) then begin
    listeEnnemis.Remove(e);
    e.fAnimationImage.Stop;
    e.fAnimationY.Stop;
    e.disposeOf;
  end;
 end;

 if listeEnnemis.Count = 0 then begin
  if FNbEnnemis < MAX_ENNEMIS then inc(FNbEnnemis);
  CreateEnnemis;
 end;
end;
```

Cette méthode **update** sera appelée dans la boucle principale que nous verrons un peu loin.
En ce qui concerne la gestion des bonus, nous allons faire très simple. Nous ferons apparaître un seul bonus à la fois à intervalle régulier (en l'occurrence, toutes les 20 secondes). Il y aura deux types de bonus : un bonus rapportant 1000 points (représenté par le logo Programmez!) et un bonus permettant de disposer d'un tir double plus puissant (représenté par un carré rouge et deux traits horizontaux jaunes). Le type de bonus sera géré aléatoirement, mais le bonus « tir double » sera plus rare que le bonus de points.
Sur la fiche principale **fMain,** nous ajoutons un **TTimer** nommé **tSpawnBonus** réglé pour se déclencher toutes les 20 secondes. Dans son événement **OnTimer**, nous générons un bonus à droite de l'écran de manière aléatoire pour déterminer :

- son type ;
- sa vitesse ;
- sa position sur l'axe Y.

Voici le code de l'événement **OnTimer** :

```
procedure TfMain.tSpawnBonusTimer(Sender: TObject);
begin
 unBonus.Parent := layZoneJeu;
 case 1 + random(5) of
  2: begin
     unBonus.Bitmap := bonusTirDouble;
     unBonus.TypeBonus := TTypeBonus.doubleTir;
     unBonus.Width := bonusTirDouble.Width;
     unBonus.Height := bonusTirDouble.Height;
    end;
  else begin
    unBonus.Bitmap := bonusPoints;
    unBonus.TypeBonus := TTypeBonus.points;
    unBonus.Width := bonusPoints.Width;
    unBonus.Height := bonusPoints.Height;
  end;
 end;

 unBonus.Position.X := jeuWidth;
 unBonus.Position.Y := random(jeuHeight-30);
 unBonus.VitesseX := 3 + random(5);
 unBonus.Visible := true;
end;
```

L'objet **unBonus** est un objet de type **TBonus** défini dans l'unité **uBonus**. Voici la définition de la classe **TBonus** héritée de **TImage** :

```
type
 TTypeBonus = (points, doubleTir);
 TBonus = class(TImage)
 private
  fSpeedX : single;
  fValeur : integer;
  fTypeBonus : TTypeBonus;
 public
  constructor Create(AOwner: TComponent);
  destructor Destroy; override;
  procedure Update;
  property VitesseX : single read fSpeedX write fSpeedX;
  property Valeur : integer read fValeur write fValeur;
  property TypeBonus : TTypeBonus read fTypeBonus write fTypeBonus;
end;
```

Nous constatons également la définition d'un enum **TTypeBonus** correspondant au type de bonus. Il n'y a que deux valeurs possibles pour cet article, mais on pourrait très bien ajouter autant de types qu'on le souhaite.
La classe **TBonus** complète le **TImage** des propriétés suivantes :

- **VitesseX** : permet de définir la vitesse sur l'axe X de déplacement du bonus ;
- **Valeur** : permet de définir une valeur pour le bonus. Dans notre cas, il s'il s'agit d'un bonus de type « points » alors la valeur contiendra le nombre de points que l'on souhaite attribuer au bonus. Nous pourrions imager un bonus de type « point de vie » qui augmenterait le nombre de points de vie du vaisseau. Pour un bonus de type « doubleTir », aucune valeur n'est nécessaire ;
- **TypeBonus** : permet de définir le type du bonus.

À ces propriétés, la classe **TBonus** ajoute également une méthode **update** qui permet de déplacer le bonus sur l'axe X. Lorsque le bonus est sorti de l'écran sur la gauche, alors on le masque et on le régénérera à droite de l'écran au bout de 20 secondes.
Le code de cette méthode est simple :

```
procedure TBonus.Update;
begin
 Position.X := Position.X - VitesseX;
 if Position.X < -witdh then begin
  visible := false;
 end;
end;
```

Voilà, nous avons à présent des vagues d'ennemis et les bonus !

La gestion des actions de l'utilisateur

Sur desktop (Windows, macOS, Linux), l'utilisateur jouera en utilisant le clavier et plus précisément les touches :

- flèche du haut : faire monter le vaisseau ;
- flèche du bas : faire descendre le vaisseau ;
- flèche de droite : déplacer le vaisseau vers la droite;
- flèche de gauche : déplacer le vaisseau vers la gauche ;
- touche CONTROL : tirer.

Pour gérer cela, nous utilisons les événements **OnKeyDown** (appuie sur une touche) et **OnKeyUp** (l'utilisateur relâche la touche) de la fiche **fMain**. Comme dans mes articles précédents, pour éviter de remplir le buffer clavier, nous passons par des booléens pour savoir si telle ou telle touche est appuyée. À chaque touche sera associé un booléen, charge ensuite à la méthode **gererTouches** de faire les actions correspondantes à l'appui sur une touche en fonction de ces booléens. Voici son code :

```
procedure TfMain.gererTouches;
begin
 if toucheAvancer then avancer;
 if toucheReculer then reculer;
 if toucheMonter then monter;
 if toucheDescendre then descendre;
 if toucheTirer then tirer;
end;
```

Cette méthode sera invoquée dans la boucle principale.

L'autre avantage de passer par des booléens est qu'il est aisé d'utiliser d'autres éléments d'interface pour réaliser ces actions. Ainsi, sur les plateformes mobiles (iOS, Android), le **layIHMMobile** sera affiché. Il contient des boutons sensibles aux pressions que le joueur exercera sur l'écran tactile. La figure 4 montre ce **layIHMMobile**.

Figure 4

Il suffit alors de positionner les booléens lors de l'appui sur ces boutons et le tour est joué : notre code fonctionnera en tactile ou au clavier en fonction de la cible choisie.

Dans la méthode **gererTouches**, nous constatons que nous appelons d'autres méthodes. Les méthodes **avancer**, **reculer**, **monter** et **descendre** permettent de déplacer le vaisseau du joueur dans la direction souhaitée. Elles sont très similaires et je ne vais détailler que la méthode **avancée** dont voici le code :

```
procedure TfMain.avancer;
begin
 if accelerationH < VITESSE_MAX then accelerationH := accelerationH + INERTIE;
 if joueur.Position.X + joueur.Width + accelerationH < jeuWidth then joueur.Position.X
 := joueur.Position.X + accelerationH;
end;
```

Cette méthode permet de faire avancer le vaisseau vers la droite de l'écran. Nous allons donc incrémenter sa position sur l'axe X d'une certaine valeur. Cette valeur que j'ai nommée **accelerationH** (pour accélération horizontale), sera la vitesse de déplacement du vaisseau avec une petite notion d'inertie : l'accélération sera faible puis va augmenter progressivement (constante INERTIE) jusqu'à atteindre une vitesse maximale (constante VITESSE_MAX).

Nous contrôlons également que la nouvelle position du vaisseau reste bien dans l'écran, le vaisseau ne devra pas sortir à droite de l'écran. Les autres méthodes **reculer**, **monter** et **descendre** sont similaires. La méthode **Tirer** est différente et va permettre de tirer un projectile.

Gestion des tirs

Le tir est un sprite (une image) qui sera créé dynamiquement à partir d'une position (l'avant du vaisseau dans notre cas) et se dirigera vers la droite de l'écran à une vitesse élevée. Si le projectile ne rencontre aucun ennemi, alors nous pourrons le détruire lorsqu'il ne sera plus visible. Si le projectile rencontre un ennemi, il sera également détruit, mais devra endommager l'ennemi. Nous verrons ce point un peu plus loin lors de la gestion des collisions.

Nous allons donc pour chaque tir effectué par le joueur créer un **TImage** contenant l'image du projectile. Cette image devra se déplacer. Comme le joueur pourra tirer des salves, nous devrons gérer une liste de tirs. Pour ce faire, nous créons une variable nommée **listeTirs** qui sera une simple liste de **TImage** :

```
listeTirs : TList<TImage>;
```

La méthode **Tirer** invoquée lorsque le joueur effectuera un tir est la suivante :

```
procedure TfMain.tirer;
begin
 if canTir then begin
  canTir := false;
  PlaySound(mpTir, mpSecours);
  if tirDouble then begin
   creerTir(joueur.Position.X + joueur.Width -12, joueur.Position.Y + joueur.Height
* 0.5 - 17 - imgTir.Height * 0.5);
   creerTir(joueur.Position.X + joueur.Width -12, joueur.Position.Y + joueur.Height
* 0.5 - 7 - imgTir.Height * 0.5);
  end else creerTir(joueur.Position.X + joueur.Width -12, joueur.Position.Y + joueur
.Height * 0.5 - 12 - imgTir.Height * 0.5);
 end;
end;
```

Le booléen **canTir** permet d'éviter que les tirs soient trop fréquents. En effet, la méthode **gererTouches** qui fait appel éventuellement à la méthode **tirer**, sera elle-même invoquée dans la boucle principale du jeu et donc exécutée très fréquemment. On rendra donc le tir possible uniquement si le booléen **canTir** est à **true**. Il est mis à **true** lorsque le joueur relâche la touche de tir.

Donc, première chose à faire dans la méthode **Tirer** est de mettre à **false** le booléen **canTir**. L'instruction suivante permet de jouer le son du tir. Nous verrons cela plus tard. Enfin, nous créons réellement le tir : soit le tir simple, soit le tir double si le joueur dispose du bonus.
Pour instancier un tir, nous appelons la méthode **creerTir** en lui passant en paramètre les coordonnées du point de départ du projectile à savoir l'avant du vaisseau :

```
procedure TfMain.creerTir(X, Y : single);
begin
 var tir := TImage.Create(nil);
 tir.Parent := layZoneJeu;
 tir.visible := false;
 tir.Position.X := X;
 tir.Position.Y := Y;
 tir.Bitmap := imgTir;
 listeTirs.Add(tir);
end;
```

La méthode **creerTir** va créer une nouvelle image à la position fournie, en affectant l'image au **layZoneJeu** et en l'ajoutant à la liste des tirs **listeTirs**.
Une dernière méthode est nécessaire à la gestion des tirs : **gererTirs**. Celle-ci va se charger pour tous les tirs présents dans **listeTirs** de gérer leurs déplacements :

```
procedure TfMain.gererTirs;
begin
 for var i in listeTirs do begin
  if i.Position.X < jeuWidth then begin
   i.Position.X := i.Position.X + VITESSE_TIR;
   i.Visible := true;
  end else begin
   listeTirs.Remove(i);
   i.Visible := false;
   i.disposeOf;
  end;
 end;
end;
```

Nous parcourons un à un les tirs présents dans **listeTirs** et nous les déplaçons sur l'axe X à la vitesse indiquée par la constante **VITESSE_TIR**. Comme indiqué précédemment, si le projectile dépasse la droite de l'écran, il n'est plus nécessaire de le gérer et nous détruisons l'objet.
Cette méthode **gererTirs** sera appelée bien sûr dans notre boucle principale.

Les collisions

Le jeu prend forme, mais il reste quelque chose d'essentiel à voir : la gestion des collisions. En effet, en l'état, nous avons un vaisseau que l'on peut diriger, il est possible de tirer, des ennemis arrivent par vagues et des bonus apparaissent, mais sans les collisions, l'intérêt est très limité !
Nous devrons gérer les collisions suivantes :

- un tir avec un ennemi ;
- le vaisseau du joueur avec un ennemi ;
- le vaisseau du joueur avec un bonus.

Pour ce faire, nous utiliserons la méthode des « boundig box » : une méthode simple, peu précise, mais suffisante pour notre jeu. Cette méthode consiste à tester si les rectangles englobants les images des sprites sont en collision ou non.
Prenons l'exemple avec deux rectangles pour lesquels nous connaissons la position de leurs origines (coin supérieur gauche), leurs largeurs et leurs hauteurs. La figure 5 matérialise cette situation.

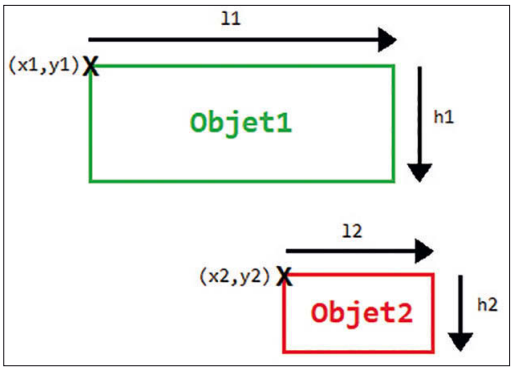


Figure 5

L'objet 1 est à la position x1,y1 (son coin supérieur gauche), il a une largeur de l1 et une hauteur de h1. L'objet 2 est à la position x2 et y2, il a une largeur de l2 et une hauteur de h2. Savoir s'il y a collision entre ces deux rectangles revient à chercher s'ils s'interceptent. Petite astuce de simplification, plutôt que de chercher les cas où les rectangles s'interceptent, cherchons les cas où ils ne s'interceptent pas.
En observant le schéma de la figure 5, nous pouvons remarquer que les rectangles ne sont pas en collision lorsque :

- x1 est supérieur à x2 + l2
- y1 est supérieur à y2 + h2
- x1 + l1 inférieur à x2
- y1 + h1 inférieur à y2

Si une de ces affirmations est fausse, cela signifie que les deux rectangles s'intersectent. C'est exactement ce que fait la méthode **enCollision** de l'unité **uMain** :

```
function TfMain.enCollision(objet1, objet2 : TControl):boolean;
begin
 result := not( (objet1.position.X > objet2.Position.X + objet2.Width) or
          (objet1.Position.Y > objet2.Position.Y + objet2.Height) or
          (objet1.position.X + objet1.Width < objet2.Position.X) or
          (objet1.Position.Y + objet1.Height < objet2.Position.Y) );
end;
```

Cette méthode prend en paramètre deux objets et renvoie un booléen à **true** lorsque les deux objets sont en collision, **false** sinon. Simple non ?
C'est effectivement simple, mais pas très précis… En effet, l'image de notre sprite est rectangulaire, mais le dessin même du sprite a une forme quelconque. La figure 6 correspond au dessin de notre vaisseau et le rectangle vert représente le rectangle du sprite.

Figure 6

Nous constatons par exemple qu'une partie du sprite est transparente en particulier le coin supérieur droit (zone en

jaune). Notre méthode de détection des collisions va détecter une collision à tort dans cette zone jaune.
Ceci est encore plus marqué avec les sprites animés, regardez les figures 7 et 8 du sprite ennemi.


Figure 7


Figure 8

Ces deux images sont à des instants différents de l'animation du sprite. Le sprite fait toujours une taille de 19x16 pixels, mais en fonction de l'étape de l'animation l'imprécision peut être importante.
Il existe d'autres méthodes pour détecter les collisions de manière plus précise, mais cela nécessiterait un article spécifique et pour notre jeu où l'action se veut rapide, nous nous satisferons de ces imprécisions.
Nous avons donc vu la méthode **enCollision** qui nous indiquera s'il y a collision entre deux objets. Or, nous avons vu qu'il nous faudra gérer les collisions de nombreux objets. C'est le rôle de la méthode **gererCollisions** :

```
procedure TfMain.gererCollisions;
begin
 for var e in ennemis.listeEnnemis do begin
  for var t in listeTirs do begin
   if enCollision(e,t) then begin
    listeTirs.Remove(t);
    t.disposeOf;
    e.PointVie := e.PointVie -1;
    if e.PointVie = 0 then begin
     Score := Score + e.Points;
     ExplosionAnim(e.Position.X, e.Position.Y);
    end else PlaySound(mpEnnemiTouche, mpSecours);
   end;
  end;
```

```
  if enCollision(joueur,e) then begin
   gameover;
  end;
 end;
```

```
if unBonus.Visible then begin
 if enCollision(joueur,unBonus) then begin
  case unBonus.TypeBonus of
   points: Score := Score + unBonus.Valeur;
   doubleTir: tirDouble := true;
  end;
  Playsound(mpBonus, mpSecours);
  unBonus.Visible := false;
 end;
end;
end;
```

Cette méthode va parcourir la liste des ennemis et pour chaque ennemi elle va tester s'il est en collision avec un tir (en parcourant la liste des tirs) et s'il est en collision avec le joueur en invoquant **enCollision**. De même, on contrôle s'il y a collision entre le vaisseau du joueur et le bonus.
En cas de collision entre un ennemi et un tir, on détruit le tir et on enlève un point de vie à l'ennemi. Si l'ennemi n'a plus de point de vie alors on incrémente le score du joueur du nombre de point associé à l'ennemi et on appelle la méthode **ExplosionAnim** qui permet de jouer une animation d'explosion et de jouer le son correspondant à l'explosion. Si l'ennemi possède encore un point de vie, alors on ne fait que jouer un son particulier (il est touché, mais pas détruit).
S'il y a collision entre un ennemi et le joueur, alors c'est le joueur qui perd et on affiche l'écran de fin de jeu (game over).
Enfin, s'il y a collision entre le joueur et un bonus, on joue un son particulier, on affecte le bonus remporté (points ou double tir) et on masque le bonus.

Les sons

Afin de plonger davantage le joueur dans le jeu, il est possible de l'enrichir de sons (tir, explosion, musique…). Delphi fournit en standard le **TMediaPlayer** qui permet de lire un fichier son ou vidéo. Les formats supportés dépendent des codecs installés sur le système cible.
Dans note projet, j'ai placé six composants **TMediaPlayer**. Il en faut plusieurs en effet pour pouvoir jouer des sons simultanément : on peut effectuer des tirs rapides tout en ayant des explosions et la musique de fond.
Il y aura donc un **TMediaPlayer** pour la musique qui tournera en boucle. À la fin du morceau, on le reprendra au début, et ce indéfiniment jusqu'à la fin du jeu.
Il y aura également quatre autres **TMediaPlayer** dédiés à chaque son : le tir, l'explosion d'un ennemi, le fait de toucher un ennemi et la récupération d'un bonus.
Enfin, il y a un dernier **TMediaPlayer** nommé **mpSecours** qui sert de **TMediaPlayer** de secours lorsqu'un même son doit être joué à plusieurs reprises rapidement et que le **TMediaPlayer** dédié au son est déjà en cours de lecture.

Dans l'unité **uSounds**, nous retrouvons trois méthodes :
- **PlaySound** : permet de lire un son. Si le **TMediaPlayer** dédié au son demandé est déjà en cours de lecture, alors le **TMediaPlayer** de secours sera utilisé. Avant de jouer le son, on s'assure de bien jouer le son depuis le début ;
- **rePlaySound** : cette méthode peut être appelée fréquemment, elle permet de rejouer un son lorsque l'on est arrivé à la fin du fichier à lire ;
- **GetAppResourcesPath** : cette fonction n'est pas spécifique pour jouer les sons, mais permet de récupérer les chemins d'accès aux fichiers packagés avec l'application en fonction des systèmes d'exploitation ciblés.

La boucle principale du jeu

L'animation **gameLoop** et plus particulièrement son événement **OnProgress** fait office de boucle principale du jeu. Comme déjà évoqué dans mes articles sur le jeu de plateforme dans le n° 247 de Programmez! ou dans le jeu de labyrinthe en 3D du n° 248, c'est dans la boucle principale que nous devons :
- gérer les actions de l'utilisateur ;
- gérer l'affichage des différents éléments du jeu (décor, sprites, bonus…);
- gérer les collisions ;
- afficher les informations (score…).

Ces actions doivent être effectuées le plus rapidement possible afin que le jeu soit fluide.

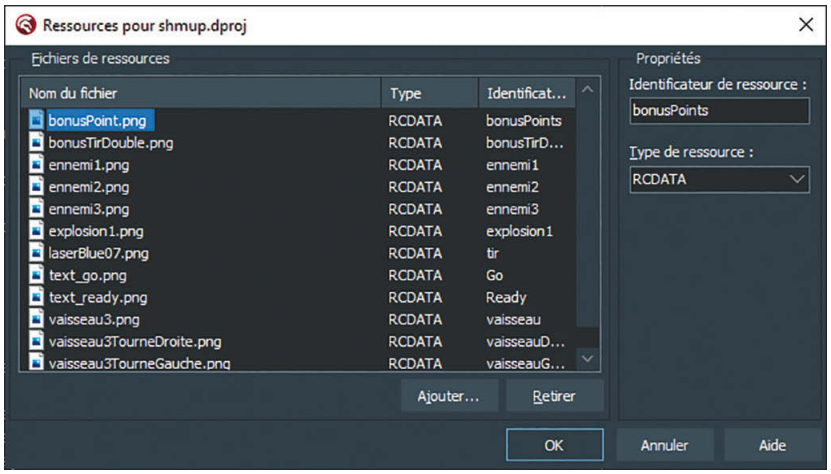


Figure 9

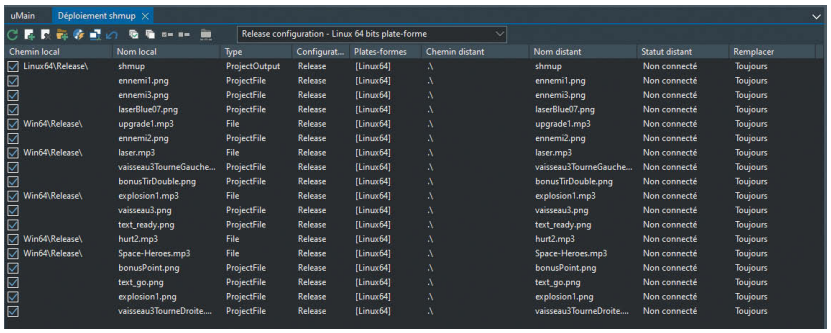


Figure 10

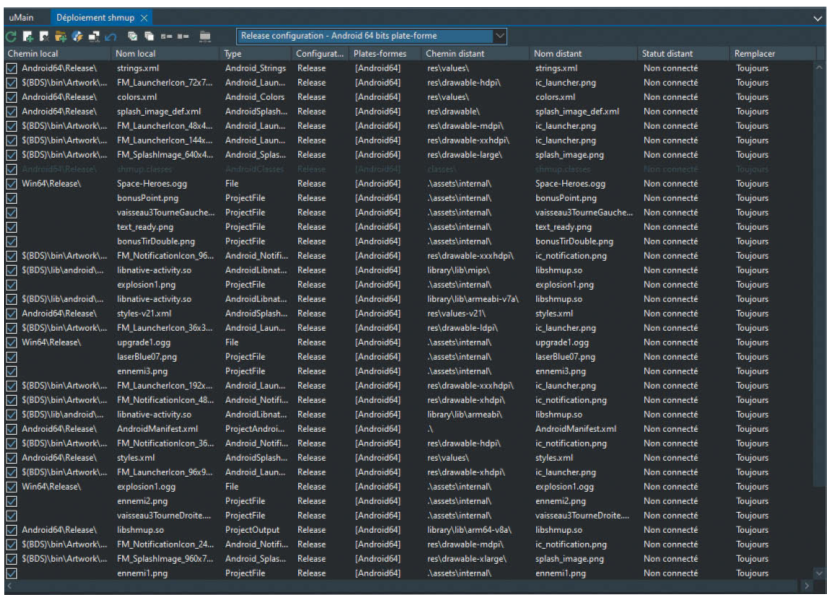


Figure 11

Avec ce que nous avons vu précédemment, notre boucle principale devient alors tout simplement :

```
procedure TfMain.gameLoopProcess(Sender: TObject);
begin
 RePlaySound(mpMusic);
 lblPoints.text := score.ToString;

 if not(estIntro) then begin
  gererTouches;
  ennemis.Update;
  unBonus.Update;
  gererTirs;
  gererCollisions;
 end;
end;
```

Le booléen **estIntro** permet de savoir si l'on est dans la phase d'introduction du jeu. Cette phase correspond à l'affichage des images « Ready » puis « Go » pendant que le vaisseau apparaît sur la gauche de l'écran.

Préparation du package final

Les images utilisées dans notre jeu sont intégrées dans les ressources du projet Delphi. Ainsi, les images seront intégrées dans l'exécutable final. Cela produit un exécutable plus gros, mais l'avantage est de ne pas avoir à fournir les images avec notre jeu. Ainsi, nous n'avons pas à nous poser les questions de savoir où sont déployées les images sur le système ciblé. Sur Desktop, nous avons accès librement au système de fichiers, mais sur les plateformes mobiles, il faut respecter les standards de la plateforme.

Pour ajouter des images aux ressources du projet sous Delphi, il faut passer par le menu « Projet/Ressources et images… », nous ajoutons les images que l'on souhaite en affectant un nom à chaque ressource (identificateur de ressource). La **figure 9** montre les ressources intégrées dans notre jeu.

Ensuite, pour charger une image depuis une ressource, on utilise la procédure **loadImageFromResource** qui prend en paramètre un **TBitmap** qui contiendra l'image lue depuis la ressource et le nom de la ressource (l'identifiant que nous lui avons donné) :

```
procedure TfMain.loadImageFromResource(var uneImage : TBitmap; resource
Name : string);
begin
 var myStream := TResourceStream.Create(HInstance, resourceName, RT_RCDATA);
 try
  uneImage.LoadFromStream(myStream);
 finally
  myStream.Free;
 end;
end;
```

Pour les sons et la musique, j'ai utilisé la fonctionnalité de déploiement fournie en standard avec Delphi. En effet, celle-ci est plus complète, car elle permet de définir où devront être stockées les ressources en fonction du système ciblé.

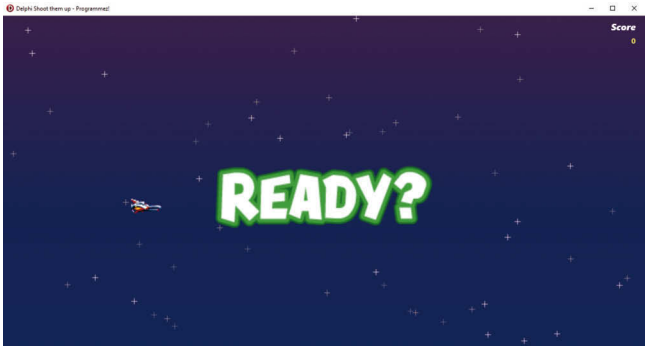


Figure 12

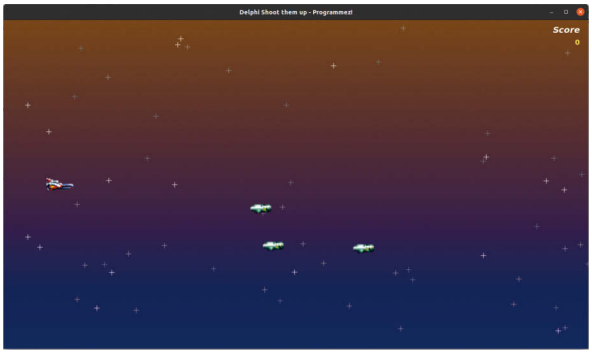


Figure 13

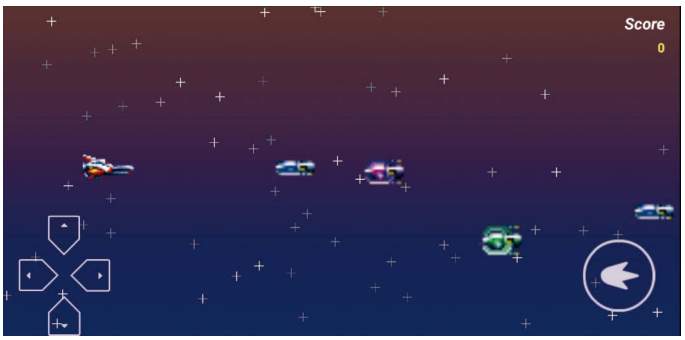


Figure 14

Pour ce faire, nous utilisons le menu « Projet/Déploiement » de Delphi. La **figure 10** montre cet écran pour la cible Linux 64 bits.

Nous y distinguons en particulier les colonnes « Chemin local » et « Nom local » qui correspondent au fichier en local sur votre poste. Les colonnes « Chemin distant » et « Nom distant » indiquent le nom complet du fichier sur la plateforme ciblée. La même fenêtre pour la configuration du déploiement sous Android est montrée à la **figure 11**.

Nous constatons qu'il y a davantage de choses. En effet, nous trouvons par exemple des images supplémentaires pour les icônes, l'écran d'accueil (splashscreen)… À noter également, j'ai utilisé des fichiers audios au format ogg sur Android alors que pour les plateformes Desktop, j'ai utilisé des fichiers au format mp3.

Nous remarquons également que sur Android, les fichiers audios seront déployés sous .**\assets\internal** alors que sur les plateformes desktop, ils seront directement dans le même dossier que l'exécutable. La fonctionnalité de déploiement permet donc d'affiner le packaging des fichiers nécessaires au fonctionnement de notre jeu.

Enfin, chose que je n'ai pas faite pour cet article, c'est également via la fonctionnalité de déploiement que nous pouvons signer avec un certificat notre application. Cela est obligatoire si vous souhaitez diffuser votre application sur les stores d'Apple, de Microsoft ou de Google. En effet, Delphi permet nativement de builder son application en vue de la diffuser sur ces stores.

Le résultat final

Le code source complet de ce projet est disponible sur mon GitHub : https://github.com/gbegreg/Shoot2D

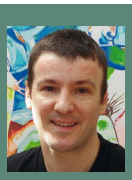
Les exécutables pour Windows, macOS, Linux et Android sont disponibles également sur mon GitHub : https://github.com/gbegreg/Shoot2D /tree/main/bin

La **figure 12** est une capture d'écran sous Windows.
La **figure 13** est une capture d'écran du jeu sous Ubuntu.
La **figure 14** montre le jeu sous Android.

Conclusion

Nous avons réalisé un petit shoot'up complet et multi-plateforme en quelques centaines de lignes de code. Certes, il n'y a pas de boss, pas une multitude de niveaux, les ennemis ne tirent pas, ce sont toujours graphiquement les mêmes ennemis, mais l'objectif était de voir les bases du codage d'un jeu en deux dimensions sans utiliser de moteur de jeu. Le fait d'utiliser un peu d'aléatoire permet de rendre chaque partie différente et le challenge du high score est là ! Les améliorations possibles sont nombreuses. À vous de jouer !

Mes débuts dans la création de jeux vidéo



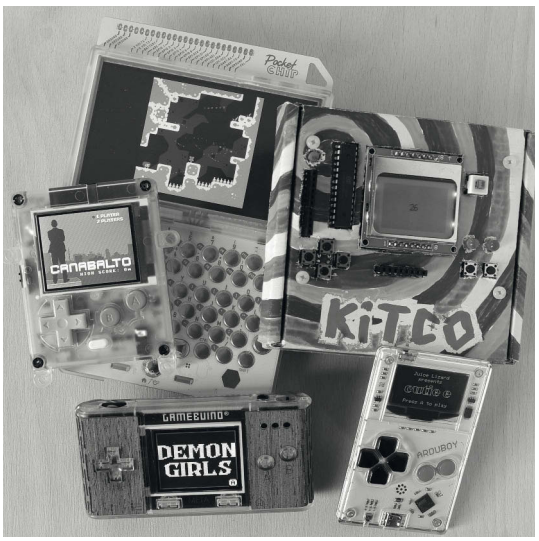
Juice Lizard

Peintre et dessinateur, créateur de petits jeux vidéo et de t-shirts, auteur du livre « Imitation de Juice Lizard ». Fan de roller, de graffiti et de philosophie. Essaye d'arrêter « Nuclear Throne ».
www.juicelizard.com

ABC DU LANGAGE C

J'était adolescent quand le premier film « Matrix » est sorti au cinéma. Cela aurait dû me motiver à entrer dans le monde magique de la programmation. Pourtant, ce n'est que récemment, après des études aux beaux-arts d'Angers pour être peintre, que je me suis plongé dans cette pratique complexe. Cela faisait longtemps que le démon du jeu vidéo m'habitait. Mais il a fallu que l'eau coule sous les ponts et que de nombreux plombiers sauvent de nombreuses princesses, pour que je franchisse la barrière de la création. J'ai appris le code avec les MOOC « ABC du langage C » et « Programmer en C » de Rémi Sharrock sur FUN MOOC. Je partais de zéro. Ce langage n'est pas réputé pour être le plus accessible, mais c'est celui qu'il vous faut pour créer des jeux sur Arduboy et Gamebuino. J'ai pu progresser ensuite grâce à l'exploration des codes des autres jeux open source, des livres sur le C et le C++, des forums de ces consoles et des visites dans des Fablabs, en région parisienne. **Figure 1**

Figure 1



Arduboy FX et Gamebuino Meta : des consoles portables pour apprendre à coder

Ces magnifiques petites bêtes s'achètent sur Internet : soit par leur campagne de financement participatif, soit sur leur site. Quelques magasins les proposent dans le monde, mais ils sont rares. Peu de personnes les connaissent, même chez les vrais gamers. Vous pourrez donc faire votre petit effet si vous en sortez une de votre poche en soirée.

La **Arduboy** a été crée par **Kevin Bates** (alias Bateske) un Américain également passionné de photo. Son écran OLED de 128 x 64 pixels affiche une image retro, mais qui se voit très bien dans le noir. Son nouveau module de mémoire flash FX permet de jouer directement à plus de 200 jeux, dès la sortie de l'emballage, via un menu intégré. Elle dispose de quatre boutons de direction et de boutons A et B. Leur design lisse et arrondi est vraiment agréable. Elle a littéralement la taille d'une carte de crédit (exceptée son épaisseur de 5 mm). Elle est solide et sa batterie a une bonne autonomie.

La **Gamebuino** Meta est française, et plus précisément de Saint Étienne. C'est **Aurélien Rodot** qui l'a conçu pour faire suite à la Gamebuino, dite Classic, qu'il avait commercialisé alors qu'il était encore étudiant ingénieur. Son écran couleurs affiche des jeux en 80 x 64 pixels par défaut. Mais si on lui demande poliment, elle ne refusera pas à fournir du 160 x 128 pixels, avec 16 couleurs indexées. Avec sa carte SD, elle peut embarquer autant de jeux qu'on le souhaite. Elle dispose d'une vraie croix directionnelle et de boutons A et B, mais également de boutons HOME et MENU (équivalent à des boutons START et SELECT). Sa forme, proche de la Gameboy Micro, est très bien conçue et sa batterie assure des parties qui peuvent durer quasiment une journée entière.

La Arduboy FX coûte **54 $** et la Gamebuino Meta **84** € (sans les frais de port). Leur site propose l'achat des consoles et accessoires, des forums, des tutoriels et bien sûr l'accès aux jeux open source créés par la communauté internationale, à consommer sans modération :
https://www.arduboy.com/
https://gamebuino.com/fr

The Guy Who Never Complains

Des débuts modestes

En bon fan de Sonic (j'avais tous ses jeux sur Megadrive) et de Mario (« Yoshi's Island » sur SNES: un bijou) je me lance dans la programmation avec l'idée de créer un jeu de plate-forme 2D. « Pas si vite, petit scarabée! » auraient pu me dire des amis professionnels du domaine. « Commence d'abord par un truc simple. Ne mets pas le « GTA » avant le « Pong ». » Mon premier programme est donc un mini livre électronique. Je me contente d'afficher du texte et de passer d'un écran à l'autre avec deux boutons. Ce qui donnera « The Guy Who Never Complains » sur Arduboy. Une lecture d'un quart d'heure, en anglais, avec ce principe: peut-on écrire quelque chose d'intéressant avec aucun élément perturbateur ? Une vie où tout va très bien (trop bien). Mais alors que l'écriture littéraire progresse, l'écriture du code butte vite sur un obstacle. Mon « mec qui ne se plaint jamais » sature déjà la mémoire de la console. Je n'ai inclus qu'un dixième du texte espéré. **Figure 2**

THE GUY WHO NEVER COMPLAINS

Figure 2

Un texte trop gros pour une console trop petite…

À cette époque, je fréquente le fablab de Malakoff (les Fabriqueurs). Et certains de leurs membres sont beaucoup plus calés que moi en informatique. Avec Papamaker (le créateur de la Kitco) nous passons une bonne demi-heure à trouver le moyen de faire rentrer ce fichu texte dans la

mémoire de la Arduboy. L'expert m'annonce que ce sera délicat, mais faisable. Il a bien réussi à intégrer de la musique dans sa version de « Tetris » sur Kitco, alors que cette machine ne pouvait pas en jouer et calculer le gameplay en même temps. Il y a de l'espoir… Surtout qu'au final, il suffisait d'ajouter trois caractères autour du texte à afficher pour le placer dans la mémoire large de la console. On l'a découvert ensemble en parcourant le site d'Arduino. Au lieu de :

```
arduboy.print("THE GUY");
```

Il faut juste écrire :

```
arduboy.print(F("THE GUY"));
```

Un « F », deux parenthèses et ça passe! Progresser en code prend du temps, mais après, ça se fait en deux secondes. Le texte est donc placé dans la mémoire programme de la Arduboy, qui est de 32 Ko, et non dans sa mémoire vive, de seulement 2,5 Ko. Elle dispose également d'un Ko d'EEPROM et peut donc gérer les sauvegardes de certains jeux. La Gamebuino Meta n'a pas ce problème et mémorise automatiquement en flash les constantes. La F() macro ne sert à rien avec elle. Ses specs sont aussi plus musclées: 256 Ko de Flash et 32 Ko de RAM pour chaque jeu.

Figure 3

```
//FRAME PERFECT
//Juice Lizard

#include "kitco.h"
#include <EEPROM.h>

byte modeJeu = 0;
#define ECRAN_SCORE 0
#define PARTIE 1

int xBalle = 82;
int xRaquette = 83;
int score = 0;
bool versLaDroite = false;
bool finPartie = false;

void setup() {

  initialiserKitco(1);
  lcdBegin();
  setContrast(60);

}

void loop() {
  switch (modeJeu) {
    case ECRAN_SCORE:
        effacerEcran(BLANC);

        ecrireLettre('0'+score%10, 41, 20, NOIR);
        if (score>=10) ecrireLettre('0'+(score/10)%10, 36, 20, NOIR);

        rafraichirEcran();
        if (toucheA()) {
          xBalle = 82;
          xRaquette = 83;
          score = 0;
          versLaDroite = false;
          finPartie = false;
          frequenceBuzzer(2000, 100);
          delai(100);
          frequenceBuzzer(1000, 100);
          delai(100);
          frequenceBuzzer(2000, 100);
          delai(100);
          modeJeu = PARTIE;
        }
        break;

    case PARTIE:
        effacerEcran(BLANC);
        ligneEcran(xBalle, 0, xBalle, 48, NOIR);
        if (versLaDroite == true) {
          xBalle = xBalle + 1;
        }
        if (versLaDroite == false) {
          xBalle = xBalle - 1;
          if (xBalle < 1) {
            versLaDroite = true;
          }
        }
        if ((versLaDroite == true) && (toucheA())) {
          xRaquette = xBalle;
          xBalle = xBalle - 1;
          xRaquette = xRaquette - 1;
          score = score + 1;
          versLaDroite = false;
          frequenceBuzzer(1000, 60);
          delai(60);
        }
        ligneEcran(xRaquette, 0, xRaquette, 48, NOIR);
        rafraichirEcran();
        if (xBalle == 0) {
          frequenceBuzzer(1000, 60);
          delai(60);
        }
        if (((xBalle >= xRaquette) && (versLaDroite == true)) || (xRaquette < 2)) {
          finPartie = true;
        }
        if (finPartie == true) {
          frequenceBuzzer(1000, 100);
          delai(100);
          frequenceBuzzer(1500, 100);
          delai(100);
          frequenceBuzzer(2000, 100);
          delai(100);
          modeJeu = ECRAN_SCORE;
        }
        break;
  }
}
```

Bip Bipbip et Frame Perfect

Maintenant il faut sortir un vrai jeu. Oui, mais le jeu le plus simple du monde! Le seul vrai jeu 1D, pour « une dimension ». Puisqu'à part le titre et le score, il ne s'affiche à l'écran que des barres verticales qui se déplacent à gauche ou à droite. Donc des graphismes en une dimension. Et le gameplay repose sur une action unique, avec un seul bouton par joueur. En une après-midi de septembre 2017, je code « Bip Bipbip », pour deux joueurs, sur Arduboy. Viendra ensuite « Frame Perfect », en solo, sur Kitco, codé en quatre heures. Le principe est simple: quand la ligne qui se déplace arrive vers ma ligne fixe, j'attends le dernier moment pour appuyer. Ma ligne se décale alors sur la position de la ligne qui se déplace, en la renvoyant dans l'autre sens. Si j'appuie trop tôt, je perds du terrain et donc me rapproche de la fin de partie. Si les deux lignes se touchent avant que j'ai appuyé, je perds directement. Le score affiche le nombre de fois où j'ai renvoyé la barre de gauche. Réflexe et stratégie dans leur forme la plus basique. **Figure 3**

Utiliser la Kitco

Cette console se trouve sur eBay à 20 €, avec piles et module FTDI inclus. Il sert à charger des programmes sur la console. Vous trouverez toutes les infos essentielles sur le site www.kitco.fr qui vous expliquera comment acheter la console, comment la souder, comment installer les logiciels et fichiers sur votre ordinateur et comment programmer des jeux (principalement avec l'IDE Arduino). Si vous cassez un des composants lors de l'étape de soudure, vous devriez pouvoir vous en procurer en notant la référence sur le site. Pour le boîtier: y'en a pas! On peut s'y prendre de différentes manières pour emballer sa console. Personnellement, j'ai peint la boîte en carton fournie avec la console, découpé les trous juste pour les boutons et les vis et fixé le tout ensemble. Pour les fonctions propres à la programmation sur cette console, elles sont formulées dans la langue de MC Solaar, et on peut les trouver sur la page GitHub de Papamaker (dans kitco/lib/src/kitco.h).
On a par exemple: toucheHaut(), toucheA(), effacerEcran(), ecrireLettre(), rafraichirEcran(), frequenceBuzzer(), etc.

Les caractéristiques techniques

- LCD monochrome rétro-éclairé (et rétro tout court!) avec une définition de 84 x 48 pixels
- six boutons d'action nus (haut, bas, gauche, droite, A et B) et un bouton ON/OFF
- un buzzer pour le son
- un microcontrôleur ATMEGA328P-PU avec 32ko de mémoire programme et 2ko de mémoire vive
- des extensions FTDI, SPI et I2C. **Figure 4**

Cutie E

Quand vous lisez « QTE », ne prononcez pas « cul-thé-euh », mais bien « Cutie E ». Car c'est bien Els Pynoo, la chanteuse du groupe Vive la fête, l'héroïne de ce quick time event musical. L'inspiration vient plus précisément du clip « You can sleep in my bed », avec également Danny Mommens et Zelda von Trapp (non, pas du jeu de Nintendo, même si elle porte parfois des cheveux roses, comme Link sur SNES).

Figure 5

Figure 4

Affichage d'un sprite

Sur Arduboy, on peut facilement afficher un sprite qui fait la taille de l'écran. Ça reste petit, mais on fait avec! **Figure 5**
J'ai affiché en fond d'écran la chanteuse allongée, en vêtements blancs moulants, sur un fond noir étoilé. Avec le passage de la photo d'origine vers le style pixelart de la Arduboy, certains ont cru voir une femme audacieusement dénudée. Ce jeu n'est donc pas inclus directement dans la Arduboy FX, pour cause de « contenu pour adultes ». Vous pouvez le charger quand même dans votre console, après l'avoir récupéré sur ma page GitHub :
https://github.com/JuiceLizard/Cutie_E

Pour créer un sprite de Arduboy, dessinez-le d'abord sur un programme de création graphique, comme Gimp, en noir et blanc, avec le bon nombre de pixels. Attention! Vérifiez que vous n'ayez que des pixels 100% blancs et 100% noirs. Si certains sont seulement à 99% noirs, vous risquez d'obtenir un sprite aberrant lorsqu'il s'affichera sur votre console. Si c'est bon, exportez-le en png. Vous devez ensuite le convertir en un bout de code. Allez par exemple sur le site « To Chars » de Crait, qui permet ce genre de conversion. Voici l'adresse:
http://www.crait.net/tochars/index.php

Chargez votre image depuis votre ordinateur vers ce site. Puis entrez ses paramètres. Ici on aura besoin que des deux premiers: la largeur du sprite et sa hauteur en pixels (donc 128 et 64 pour cet exemple). Cliquez en bas sur le bouton rouge « Convert ». Votre sprite devrait s'afficher sur la page, et juste en dessous, le morceau de code correspondant. Maintenant, copiez-collez le résultat depuis le site vers votre code complet, dans la fenêtre de l'IDE Arduino, en dehors de void setup() et void loop(). Cela fera une très grande ligne, qui dépassera de la fenêtre. Ce n'est pas fini! On doit encadrer correctement ces données pour que la Arduboy les comprenne bien.
// On ajoute sur la ligne d'avant :

```
const unsigned char Els[] PROGMEM ={
```

// Le début de votre copier-coller ressemble à ça :

```
0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x10, //…
```

// Et sur la ligne d'après :

```
};
```

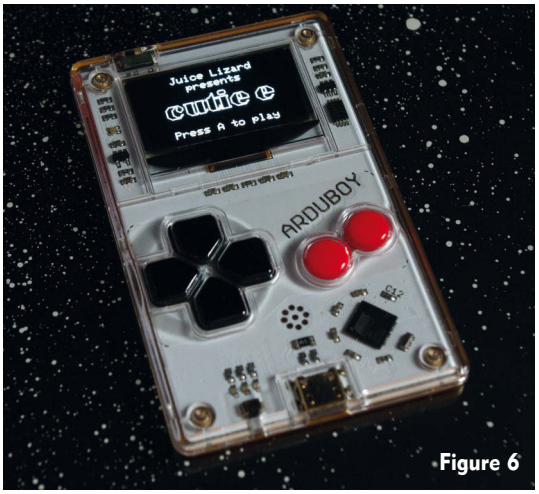


Figure 6

Tout ceci indique que le sprite est enregistré dans la mémoire programme de la console, sous forme de tableau nommé « Els ». Et n'oubliez pas le point-virgule tout à la fin, sinon ça ne marchera pas!
Nous avons également besoin d'invoquer la fonction arduboy.clear() au début du void loop() et arduboy.display() à la fin de ce même void loop() pour effacer tout l'écran et pour afficher ensuite tout ce qui aura été stocké dans le buffer. Ceci répété à chaque frame.
Une autre fonction propre à la console sert à afficher le sprite voulu au bon endroit :

```
arduboy.drawBitmap(0, 0, Els, 128, 64, WHITE);
```

Nous avons en paramètres de cette fonction, dans l'ordre: deux zéros séparés d'une virgule, pour la position en « x » et en « y » du pixel supérieur gauche du sprite. (Les « x » partent de la gauche de l'écran et vont vers la droite et les « y » partent du haut et vont vers le bas). Le nom du sprite (ici « Els », c'est le prénom de la chanteuse). Puis la largeur et la hauteur du sprite en pixels (128 et 64). Et pour finir, la couleur du sprite (ici « WHITE » pour blanc). Sur Arduboy, l'affichage des sprites se fait par défaut en pixels blancs et en pixels transparents, ou bien en pixels noirs et en pixels transparents.
Pour faire bouger un sprite, placez des variables de type « int » au niveau des deux premiers paramètres de cette fonction (les « x » et les « y »). Puis modifiez les valeurs de ces variables ailleurs dans le code, en fonction du déroulement de votre jeu. Si vous décidez que votre sprite monte à vitesse constante sur l'écran, codez pour que son « y » commence par valoir 50, par exemple, puis diminue de 1 à chaque frame.
Ouf! Voilà… On y est. Évitez néanmoins de montrer ce genre de pixels érotiques à vos enfants. Leur âme pourrait être corrompue de manière irréversible! **Figure 6**

Les boutons

Bon, je ne voudrais pas vous mettre la pression, mais il y a des boutons qui attendent d'être appuyés! Tout est prévu, grâce à la fonction pollButtons() qu'on appelle au début de chaque frame. Elle vérifie la pression de chaque bouton et garde en mémoire leur état lors du précédent appel. Utile pour servir les fonctions justPressed() et justReleased() qui renvoient la valeur vrai quand le bouton indiqué en paramètre vient d'être appuyé, ou vient d'être relâché. La fonc-

tion pressed() vérifie seulement si un bouton est appuyé, mais peut prendre en compte plusieurs boutons à la fois. Elle renvoie vrai si tous les boutons indiqués en paramètres sont appuyés au moment de son appel.
On peut par exemple coder l'utilisation d'un bouton de cette manière :

```
if(arduboy.justPressed(A_BUTTON)){
// si le bouton A vient d'être appuyé
    gamestate = 1; // passe de l'écran titre au jeu
    randomSeed(millis()); // calibre l'aléatoire
    fromZeroToFive = random(6); // choisit le premier bouton qui sera affiché
    while (arduboy.buttonsState()) {}
// attend que tous les boutons soient relâchés
  }
```

Dernière astuce pour se souvenir lequel des deux boutons rouges est le A et le B: ils sont placés juste en dessous des lettres dorées correspondantes du mot « ARDUBOY » sur la face avant de la console (tout est pensé, tout est millimétré!).

La musique

Si ce n'est pas déjà fait, je vous conseille vivement d'écouter et de voir le clip de Vive la fête « You can sleep in my bed ». Vous serez transportés. Ensuite, il faut télécharger la bibliothèque ArduboyTones dans votre Arduino IDE. Vous trouverez des informations détaillées sur GitHub à propos de cette bibliothèque :
https://github.com/MLXXXp/ArduboyTones

Il donc est conseillé d'inclure en début de code :

```
#include <Arduboy2.h>
#include <ArduboyTones.h>

Arduboy2 arduboy;
ArduboyTones sound(arduboy.audio.enabled);
```

Après ça, essayez tant bien que mal d'adapter comme moi la mélodie originale pour qu'elle rentre dans la console 8-bit et son buzzer quatre canaux. Il faut enregistrer votre mélodie nommée « YouCanSleep » dans la mémoire programme de votre console. Ça pourrait donner ceci :

```
const uint16_t YouCanSleep[] PROGMEM = {
NOTE_G3H,160, NOTE_REST,90, NOTE_G3H,160, NOTE_REST,90,
NOTE_B3H,160, NOTE_REST,340, NOTE_G3H,250, NOTE_REST,100,
NOTE_D3,250, NOTE_REST,150, NOTE_D3,250, NOTE_REST, 50,
 TONES_REPEAT
};
```

L'essentiel des données est constitué de paires de fréquences, en Herz, et de durées, en millisecondes, pour former chaque note musicale. Mais vous voyez qu'ici les fréquences sont codées sous un format en lettres :
NOTE_<lettre><S optionnel pour dièse><numéro de l'octave><H optionnel pour un volume plus élevé>
La lettre au début est la notation anglaise de la note. Voici la correspondance:
A = la, B = si, C = do, D = ré, E = mi, F = fa et G = sol.
NOTE_REST correspond à un silence (utile pour bien séparer deux notes consécutives). TONES_REPEAT placé à la fin indique que la mélodie sera jouée en boucle. Un TONES_END aurait indiqué à la console qu'il est temps que la musique s'arrête avant que vos voisins ne se plaignent des bip bip. Votre séquence de sons est bien enregistrée. Pour la lancer pendant le jeu, il faudra activer cette fonction à la bonne frame :

```
sound.tones(YouCanSleep);
```

Tout est maintenant bien calibré pour nos oreilles. Et pour le nez, est-ce que c'est bien carré?

Square Nose

J'ai pris une bonne résolution

La Arduboy ne dispose que d'un mode d'affichage en 128 x 64 pixels. Mais « Square Nose » a été conçu dès le départ avec des graphismes en 64 x 32 pixels. Tous ses sprites sont constitués de « gros pixels » (2 x 2 pixels de l'écran). Même si la physique du jeu reste bien en 128 x 64 pixels. Cela permet d'obtenir des graphismes plus simples et plus lisibles, et d'affirmer le style pixel art minimaliste. **Figure 7**

Va, cours saute et nous mange!

Lorsqu'on prononce « Sonic » à l'envers, ça donne « qui n'ose ». On est proche de « Square Nose », l'écureuil héros de cette aventure au nez carré (car pixelisé). Ce remake du hérisson bleu est d'un minimalisme radical. Je trouve qu'il possède au moins deux avantages par rapport à son modèle: le personnage se déplace sans inertie latérale (dès que vous appuyez sur gauche ou droite, il atteint sa vitesse maximale) et aucun élément susceptible d'interactions ne surgit de l'extérieur de l'écran (pas de scrolling, aucun risque qu'un ennemi vous surprenne si vous vous déplacez vite).
L'histoire: vous êtes capturé par le docteur Meggan, qui vous utilise dans son laboratoire pour tester des recettes de croquettes pour chiens et chats. Vous êtes chronométré et un compteur affiche le nombre de croquettes que vous avez englouti. L'entreprise « True Pasta » sera intéressée pour savoir quelle recette de croquettes est la plus irrésistible, et donc la plus rentable. **Figure 8**

Black-blanc-blink

Le compteur de croquettes mangées s'affiche en fond d'écran, dans une sorte de gris étrange. Tout comme le « O » de « SQUARE NOSE » à l'écran-titre, le numéro du niveau et le « GAME OVER ». L'écran de la Arduboy n'affiche que des pixels de lumière blanche sur fond noir. Mais avec un maximum de soixante images par seconde et en faisant clignoter ces pixels, on arrive plus ou moins à obtenir du gris. Ainsi je code le compteur de croquettes mangées en affichant les sprites une fois sur deux :

```
// draw the number of collected meaty rings
   if (arduboy.everyXFrames(2)) {
// cette fonction retourne vrai toutes les deux frames
     drawNumber(collectedMeatyRings / 10, 46, 24);
// on affiche le chiffre des dizaines grâce à la division entière par dix
     drawNumber(collectedMeatyRings % 10, 66, 24);
// on affiche le chiffre des unités grâce au reste de la division par dix
   }
```

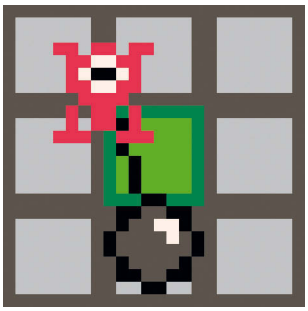


Figure 10

Figure 11

Figure 7

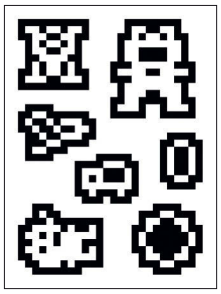


Figure 8

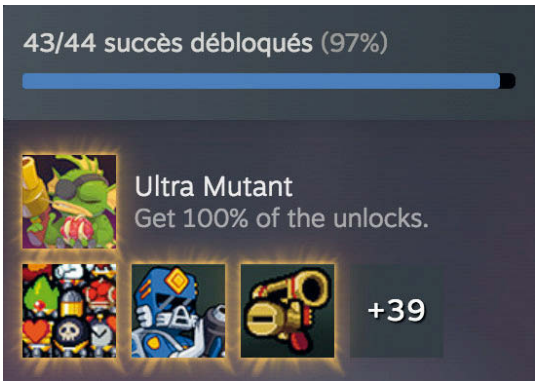


Figure 12

Si vous souhaitez porter « Square Nose » sur Playdate, vous opterez sans doute pour des gris tramés (alternance de pixels blancs et noirs sur l'espace de l'écran) au lieu de faire clignoter quoi que ce soit. Car cette autre petite machine possède une résolution bien plus élevée: 400 x 240 pixels, au lieu de 128 x 64 pixels pour la Arduboy. Sachez également que le hardware de la Arduboy est open source, et pas uniquement les jeux, il est donc possible de fabriquer votre propre version de cette console. **Figure 9**

Seul, on est peu de chose

Square Nose ne va pas rester seul longtemps dans cette galère. Dans le premier tableau, il rencontre Bug l'insecte et Bird l'oiseau. Leurs mouvements sont rectilignes et horizontaux. Au deuxième tableau, un autre animal a la faculté de sauter comme lui, c'est Frog la grenouille. Au troisième et dernier tableau, on tente d'esquiver Squiddy et son boulet, qui se déplacent en trajectoires circulaires l'un autour de l'autre, en alternance. Ce dernier personnage est le seul qui provient d'un autre jeu: « Squiddy » créé par Aphrodite sur Pico-8 et porté par catsfolly sur Pokitto. C'est un très bon petit jeu à un seul bouton, avec un concept simple mais original. Voici le lien vers sa page Pico-8: **Figure 10**
https://www.lexaloffle.com/bbs/?tid=28824

Petit à petit…

Pico-8 est une console virtuelle déjà très appréciée, dont le nom vient de la phrase « Pico Pico » en japonais. Ceci veut dire pixelisé ou qui base son style sur des limitations techniques (« low-fi »). Pico est aussi un préfixe scientifique lié à une unité, qui vient de piccolo ou « petit » en italien, et qui divise cette unité par dix puissance douze. Les jeux Pico-8 s'affichent en 128 x 128 pixels, parmi une palette de 16 couleurs. Et ils ne pèsent pas plus de 32 Ko. Ils sont codés en Lua et le logiciel Pico-8 contient tout ce qu'il faut pour les créer

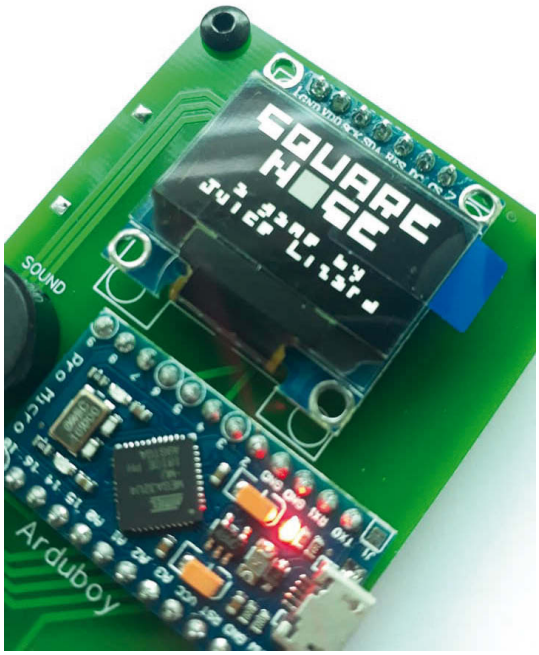


Figure 9

et les utiliser: jeu, code, sprites, maps, sons et musiques.
Le site de Pico-8 où vous pouvez directement jouer à des jeux, même sans acheter le logiciel :
https://www.lexaloffle.com/pico-8.php

Pokitto est une console similaire à la Gamebuino Meta, mais avec la forme d'un petit bonhomme et un écran légèrement plus grand: 220 x 176 pixels au lieu de 160 x 128 pixels. Une fois commandée sur ce site :
https://www.pokitto.com/shop/
Il ne sera pas nécessaire de la souder, comme la Kitco, mais il faudra la monter, notamment avec des pièces de Lego Technic. Pas très chère (environ 50 euros), fabriquée en Europe, et terriblement ludique! Cette console en couleurs vous permettra de créer des jeux en codant principalement en C++. Chaque jeu tient sur un mémoire programme maximum de 256 Ko, 36 Ko de mémoire vive et 4 Ko pour des sauvegardes. Le tout livré avec une carte micro SD, avec de nombreux jeux déjà inclus. La console intègre un menu de sélection des jeux, comme sur la Gamebuino Meta. Le nom Pokitto ressemble à « pocket kid » et à « Poquito », qui signifie « peu » ou « petit » en espagnol. En japonais, Pokitto veut aussi dire « le son d'un petit bâton qui se casse ». Mais à moins d'utiliser une batte de baseball, votre Pokitto ne risque pas de se briser! **Figure 11**

On veut des couleurs! Mon passage sur la Gamebuino Meta.

Mon ancien MacBook n'est pas assez récent pour recevoir une version de l'IDE Arduino qui peut coder pour la Gamebuino Meta. L'achat d'un MacBook Pro d'occasion s'impose. Une évolution assez judicieuse car l'écran plus grand et aux couleurs plus précises est un atout pour mon activité de peintre. La puissance supplémentaire me permet aussi d'avoir accès à plus de jeux sur Steam. « Celeste », « CupHead », « DiscRoom », « Gris », « Hollow Knight », « Later Alligator », « Lethal League », « Nuclear Throne »… Tant de jeux à jouer, et si peu de temps! **Figure 12**

Hello Gamebuino

Voici le code pour afficher le fameux « hello, world » sur votre Gamebuino Meta :

```
#include <Gamebuino-Meta.h>

void setup() {
 gb.begin();
}

void loop() {
 while(!gb.update());
 gb.display.clear();

 gb.display.print(« hello,world »);
}
```

Pour le faire fonctionner, vous devez installer une version de l'IDE Arduino récente. Allez ensuite dans son gestionnaire de cartes pour installer les cartes « Arduino SAMD » et « Gamebuino Meta ». Puis, dans la partie outils et cartes, sélectionnez « Gamebuino Meta ». Dans la partie croquis/inclure une bibliothèque/gérer les bibliothèques, cherchez « Gamebuino Meta » et installez-la. Allumez votre console portable française open source préférée et branchez-la à votre ordinateur avec un câble USB. Allez dans outils/port et trouvez celui de votre console (peut-être « Arduino/Genuino Zero (Native USB Port »)). Sur certains Windows, il est nécessaire d'installer des drivers. Cliquez sur la flèche verte en haut à gauche de la fenêtre de votre code pour le téléverser (ou bien allez dans croquis/téléverser).
Si je ne suis pas assez clair, retrouvez toutes les infos sur les tutoriels du site Gamebuino : https://gamebuino.com/fr/academy

Square Nose Color

À force de faire des bonds, Square Nose a sauté d'une console à l'autre. Dans cette version Gamebuino Meta, vous trouverez tout de la version Arduboy, mais avec quelques ajouts. Les couleurs bien sûr. Un chronomètre visible en bas de l'écran (il faut bien s'adapter au nouveau format). Un décor différent pour chaque tableau. Ils sont inspirés de zones industrielles, comme dans le jeu « Portal », et sont codés avec des formes géométriques colorées et des doubles boucles « for » (pour les répéter en rangées et en colonnes). Le docteur Meggan est un peu plus présente. On voit sa main fondre sur le pauvre écureuil et le capturer inévitablement, dans une courte séquence d'introduction. Tout ceci est fort plaisant à l'œil. Et si on en capturait un bout pour le montrer sur Internet?

Créer un GIF depuis un jeu Gamebuino Meta avec Piskel

Capture depuis la console

Allumez votre console et sélectionnez le jeu dans lequel vous souhaitez faire votre capture. Jouez jusqu'à arriver à l'endroit où vous souhaitez démarrer l'enregistrement. Appuyez sur le bouton HOME, puis sélectionnez « ENRG VIDEO » (pour l'ancienne version) ou l'icône en forme de caméra (pour la version récente) et appuyez sur A. La partie va reprendre (un peu plus lentement, car la console aura du mal à tout calculer en même temps) et pendant que vous jouez, 75 images vont être enregistrées à la suite (l'écran entier). Si le framerate est par défaut à 25 images par seconde, cela fera trois secondes de vidéo. Le son n'est pas pris en compte. Le jeu est à nouveau stoppé, pour laisser le temps à la console de convertir les données et de bien les enregistrer sur la carte micro SD. Ensuite, vous avez le choix entre reprendre votre partie normalement ou relancer une capture vidéo. Vous pouvez donc les enchaîner pour obtenir des vidéos plus longues.

Transfert vers l'ordinateur

Éteignez votre console et retirez la carte micro SD. Branchez-la à votre ordinateur grâce à l'adaptateur USB fourni avec la Gamebuino Meta. Ouvrez le dossier correspondant au jeu sur lequel vous avez fait votre capture. Ouvrez le dossier « REC ». Récupérez le dernier fichier en BMP pour le mettre sur votre ordinateur. Son icône devrait ressembler à une ligne verticale. En fait, il est encore sous la forme d'une seule image où les 75 écrans capturés sont collés les uns au-dessous des autres.

Transfert vers le logiciel Piskel

Maintenant, démarrez le logiciel Piskel. Il est gratuit et disponible en version offline téléchargeable, pour Windows, macOS et Linux. Faites glisser votre fichier BMP de votre bureau vers la fenêtre Piskel. Une fenêtre « Import and Merge » s'ouvre. Cliquez sur « Import as spritesheet » et ajustez « Frame size » à 80 x 64, au lieu de 80 x 4800 (sauf si votre jeu est codé en 160 x 128 pixels). Cliquez sur « import » puis sur OK. Au milieu en grand s'affiche la frame sélectionnée, que vous pouvez modifier à loisir. Sur la colonne de gauche se déroulent toutes les frames qui composent votre animation. En haut à droite s'affiche un petit aperçu de l'animation qui tourne en boucle. Elle est réglée par défaut à 12 FPS. Vous pouvez la monter à 24 FPS (le maximum) car c'est la valeur la plus proche pour les jeux Gamebuino Meta, par défaut. **Figure 13**

Exporter sous forme de GIF

Cliquez sur l'icône blanche en forme de paysage (une montagne et un Soleil dans un carré), tout à droite de la fenêtre Piskel. C'est la partie qui sert à exporter. Cliquez sur « GIF ». Vous pouvez redimensionner votre GIF en déplaçant le curseur du « Scale » qui se trouve en haut (jusqu'à 32.0x). La

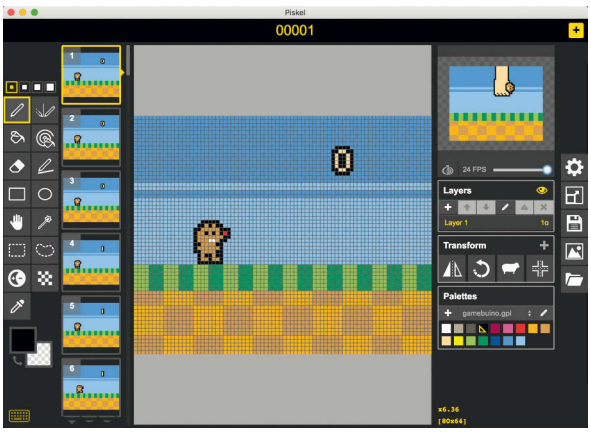


Figure 13

Figure 14

Figure 15

résolution correspondante s'affichera automatiquement juste en dessous. Cliquez ensuite sur le bouton jaune « Download ». Patientez quelques secondes pour que le GIF soit créé. Donnez ensuite un titre à votre GIF, choisissez l'endroit où il sera sauvegardé et cliquez sur « Save ». Et voilà! Votre GIF est créé. Si mes explications sont insuffisantes, peut-être trouverez-vous de l'aide en vous connectant au forum ou au Discord de Gamebuino. Nous ne manquons pas d'âmes charitables prêtes à venir au secours des plus démunis.

(BONUS) La palette Gamebuino Meta

Pour modifier des images de Gamebuino Meta ou pour créer des sprites pour cette console, vous aurez probablement besoin d'importer la palette de 16 couleurs indexées de la console vers le logiciel Piskel. Récupérez le fichier gamebuino.gpl sur le site de la console, il fait environ 400 octets. Vous le trouverez en tapant « palette Gamebuino Piskel » dans un moteur de recherche classique, hors du site Gamebuino (le moteur de recherche intégré au site de la console n'est pas hyper au point). Puis, dans Piskel, cliquez sur le signe « plus » blanc dans la partie « Palettes » en bas à droite. Une fenêtre « Create Palette » s'ouvre. Cliquez sur le bouton jaune « Import from file ». Sélectionnez votre fichier gamebuino.gpl et ouvrez-le. Cliquez sur « Save » en bas à droite. Vous voici donc en possession d'un blanc, deux gris un peu chauds, un noir, un violet qui ressemble plus à un rouge foncé, un rose, un rouge, un orange clair, un marron clair, un beige, un jaune, deux verts et trois bleus. Plus qu'il n'en faut Picasso!

Dumbulance

Game jam

Les pros conseillent souvent aux novices de participer à des game jam pour progresser. La Ludum Dare est une des plus connue. C'est la seule à laquelle j'ai participé pour l'instant. Elle est régulière et en ligne. L'édition numéro 46 s'est déroulée autour du thème « Keep it alive ». Ma proposition fut un nouveau petit jeu sur Gamebuino Meta: « Dumbulance ». Je ne me doutais pas qu'au même moment les ambulances, pas si bêtes, et les soignants en général deviendraient si importants en cette période de pandémie. **Figure 14**

« Une ambulance fonce vers l'hôpital le plus proche pour sauver des vies. Mais son conducteur est soûl. Aidez les animaux qui se trouvent sur la route à éviter le véhicule fou. » Ainsi est décrit le jeu sur sa page Ludum Dare, à retrouver ici : https://ldjam.com/events/ludum-dare/46/dumbulance

Je remercie les autres participants d'avoir été si bienveillants dans leurs commentaires, face à ce jeu ultra basique. Les graphismes mignons et drôles ont été appréciés, ainsi que le twist du gameplay. Puisque vous ne contrôlez pas l'ambulance, mais les animaux qui arrivent au hazard sur son chemin. Et pour les éviter, une simple pression sur haut ou bas suffit.

Avec le peu de temps imparti (trois jours pour la catégorie « jam »), la décision était prise de réutiliser les personnages de mon précédents jeu « Square Nose Color ». Pauvres petites bêtes recouvertes de pixels! Après avoir vécu dans un laboratoire, elles risquent leur vie sur les routes de France. Heureusement que le code est là pour nous faire oublier la dureté de la vie. « Coder pour oublier, mais ne pas oublier de coder. »

L'émulateur en ligne de la Gamebuino Meta

Si les nombreux participants de cette game jam ont pu noter mon jeu, sans posséder le précieux hardware, c'est en partie grâce à l'émulateur en ligne de la console. On peut essayer les jeux publiés sur le site de Gamebuino, directement sur leur page de créations. Le son est absent, le framerate n'est pas au mieux de sa forme, mais en attendant la console, qu'on vient évidemment de commander, ce sera déjà suffisant. On peut également se diriger vers un émulateur externe en ligne, que voici : http://games.aoneill.com/wasm-proto/

Il tourne assez bien au niveau de l'image. Le son est pris en compte, même s'il tourne parfois au ralenti. Il arrive que le jeu freeze, mais globalement c'est du bon matos. Cette page propose directement une dizaine de titres de la Gamebuino Meta, dont le très mignon plateformer « Cats and Coins » de Siegfried Croes. Mais pour le même prix (c'est à dire zéro euros) vous pouvez charger un autre jeu sous le format « .bin » assez facilement et le tester deux secondes plus tard. Vous n'avez donc plus aucune excuse pour ne pas vous lancer dans la création du prochain « Flappy Bird » sur Gamebuino Meta! Let's go! **Figure 15**

Demon Girls

En 2020 sortait sur Steam un jeu gratuit, court, mais avec un succès incontestable: « Helltaker » par VanRipper. Ce puzzle game des enfers, rempli de waifus démoniaques, allait inspirer bon nombre de fanarts et de détournements, parfois très chauds, à partager sans modération sur les Discord, Reddit et autres. J'ai moi-même beaucoup accroché, et je décidais de poursuivre l'expérience en proposant mon demake sur

Gamebuino Meta. Comme le gameplay est très proche d'un « Sokoban », j'ai pu démarrer le code en me servant du chapitre « Création d'un jeu Sokoban » sur Kitco, dans le livre « Construisez et programmez votre console de jeux open source » par Audric Gueidan (2020, édition Dunod).
Figure 16

Le son

La Gamebuino Meta est limitée par ses musiques 8-bit, mais possède tout de même des atouts, avec un haut-parleur, une prise jack et une carte micro SD, qui peut contenir de lourds fichiers musicaux. L'ébauche du jeu n'exploitait que les trois sons de bases proposés par la Gamebuino Meta, à savoir **gb.sound.playTick()**, **gb.sound.playOK()** et **gb.sound.playCancel()**. Mais je sentais qu'il y avait du potentiel pour faire beaucoup mieux. Surtout que dans le « Helltaker » d'origine, les musiques de Mittsies sont géniales. On peut les écouter gratuitement ici : https://mittsies.bandcamp.com/

Les fichiers sons, en WAV 8-bit, stockés dans la carte micro SD, peuvent être lus pendant le jeu, grâce à la fonction **gb.sound.play()**. Il faut juste s'assurer de les lancer à un moment où le jeu ne ralentit pas, sinon ça s'entend. Malheureusement, toutes les fonctions d'affichage de la console ne sont pas optimisées de la même manière. Il faut tester pour voir ce qui est le moins gourmand en calcul, entre les fonctions d'affichage de formes géométriques colorées, les affichages de sprites simples et les affichages de sprites avec déformations. La fonction gb.display.drawImage() par exemple, peut accueillir deux paramètres facultatifs de mise à l'échelle. Utile pour modifier la taille, la proportion ou l'orientation d'un sprite facilement, mais ce n'est pas forcément ce qu'il y a de plus optimisé.

Befriend all demons

Si vous trouvez les trois symboles cachés dans la partie Enfer de mon petit jeu, vous accédez à une chanson bonus. Elle n'est pas présente dans le « Helltaker » original. C'est une parodie de « Bad » de Michael Jackson, que j'ai nommé « Befriend all demons ». J'ai réécrit les paroles et demandé à un chanteur professionnel de la concrétiser. J'ai utilisé le site www.fiverr.com pour contacter ce prestataire. Le fichier n'est pas disponible sur ma page GitHub à l'heure où j'écris ces lignes. Mais c'est prévu, ainsi qu'une autre chanson en français dans la seconde partie du jeu, qui se déroule au Paradis. Abonnez-vous à ma chaîne YouTube pour être sûrs de ne rien rater! **Figure 17**

Les bruitages

Pour créer les différents effets sonores du jeu, le mieux est d'utiliser la fonction **gb.sound.fx()**. Elle accepte sept paramètres. La forme de l'onde: NOISE ou SQUARE. Un booléen pour indiquer si la note est la dernière du bruitage (« un » par défaut et « zéro » pour indiquer le dernier son). Le volume en début de note. La variation du volume en cours de note. La variation de la fréquence de la note (si elle devient plus aiguë ou plus grave). La fréquence en début de note. Et enfin la durée de la note (la valeur indiquée équivaut à celle en millisecondes, divisée par vingt).

Voici par exemple comment est mise en mémoire le couinement de Square Nose, le nez-cureuil, dans le jeu « Demon Girls » (il apparaît vers la fin, dans une cinématique) :

```
const Gamebuino_Meta::Sound_FX squirrel[] = {
  {Gamebuino_Meta::Sound_FX_Wave::SQUARE, 0, 80, 0, -16, 32, 10}
};
```

Cela donne un effet composé d'un seul son, à volume constant, qui démarre aiguë et qui finit encore plus aiguë (les bestioles de jeux vidéo sont souvent insupportables!), et qui dure 200 millisecondes. Un programme vous permet de créer et de tester directement les bruitages depuis votre Gamebuino Meta, c'est « GSFX GUI ». Vous pourrez trouver plus d'informations à cette adresse :
https://gamebuino.com/fr/academy/reference/gb-sound-fx

L'intro

En démarrant le jeu, on peut voir mon logo de tête de blaireau pixelisé s'agiter sur fond noir et se stabiliser progressivement sur le centre de l'écran. Je souhaitais l'accompagner d'un effet sonore original, mais je n'avais pas d'idée de mélodie. J'ai donc utilisé la fonction **gb.sound.tone()** qui prend deux paramètres: la fréquence en Herz de la note et la durée en millisecondes. Cette fois-ci, on peut placer des variables comme paramètres. J'ai alors obtenu assez rapidement un

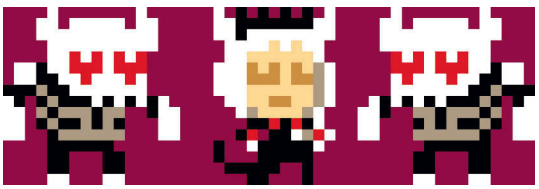


Figure 16

Figure 17

effet sonore intéressant, qui collait bien à mon animation visuelle.

```
  xLogo = random(-logoTrouble, logoTrouble);
  yLogo = random(-logoTrouble, logoTrouble);

// intro sound
  gb.sound.tone(abs(xLogo*yLogo), 80);
```

Ici, la variable logoTrouble va de 120 à 0, puis de 0 à 120, lors de l'animation d'intro. L'enchainement de bips se fait alors globalement de l'aiguë au grave, puis du grave à l'aiguë, avec une part d'aléatoire.

Comment créer vos propres niveaux!

Pour laisser parler tous vos talents de level designer, et parce que « Mario Maker » c'est bien, mais on ne peut pas draguer des jolies filles démoniaques, vous aurez besoin de récupérer le code de « Demon Girls » et de l'arranger à votre sauce. Vous le trouverez sur ma page GitHub :
https://github.com/JuiceLizard/DemonGirls

Ouvrez-le sur l'IDE Arduino et sélectionnez l'onglet level.h pour trouver les tableaux de caractères à deux dimensions qui constituent l'agencement des niveaux. Ailleurs dans le code, ces tableaux sont lus et les éléments affichés grâce à une double boucle « for » (une pour les rangés et une pour les colonnes). Chaque écran de jeu est composé d'une grille de cases, qui font chacune 16 x 16 pixels. Les commentaires inclus dans le code indiquent la correspondance entre les éléments du jeu et les caractères du tableau ('M' pour un mur infranchissable, '@' pour le héros super classe, 'K' pour une clé en or massif, 'D' pour une fille démon bien habillée, etc.). Ne touchez pas au premier tableau de variables nommé « level », il ne sert qu'à recevoir les copies des autres tableaux de niveaux en données constantes, et aussi à être modifié par le programme pendant que vous jouez. Si vous bloquez dès le niveau 1, car vous n'êtes définitivement pas fait pour ces foutus puzzles où il faut réfléchir (ça arrive), prenez ce code et changez les deux caractères 'M' qui sont entre le joueur '@' et la fin du niveau 'D', et transformez-les en espaces vides ' '. Puis rechargez le jeu. Vous aurez créez un passage fort avantageux entre le héros et la première fille démon fatiguée.

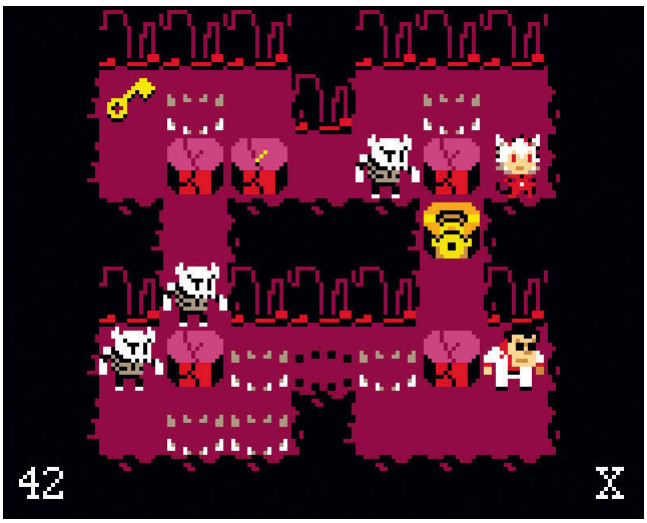


Figure 18

```
// level 1 Pandemonica
const char level1[NB_ROWS_LEVELS][NB_COLUMNS_LEVELS] =
{{'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'},
 {'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'},
 {'M', 'M', 'M', 'M', 'M', 'M', ' ', ' ', '@', 'M', 'M', 'M'},
 {'M', 'M', 'M', ' ', ' ', ' ', 'S', ' ', ' ', ' ', 'M', 'M',
 {'M', 'M', 'M', ' ', ' ', ' ', 'S', ' ', ' ', ' ', 'M', 'M', 'M'},
 {'M', 'M', 'M', ' ', 'S', ' ', ' ', 'S', 'M', 'M', 'M', 'M'},
 {'M', 'M', ' ', ' ', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
 {'M', 'M', ' ', ' ', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'},
 {'M', 'M', ' ', 'B', ' ', ' ', 'B', ' ', ' ', 'M', 'M', 'M'},
 {'M', 'M', ' ', 'B', ' ', 'B', ' ', ' ', ' ', 'D', 'M',
 {'M', 'M', ' ', 'B', ' ', 'B', ' ', ' ', ' ', 'D', 'M', 'M'},
 {'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'},
 {'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
 {'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'} };
```

Voilà comment vous pouvez modifier le level design. Pour les graphismes, votre serviteur a tout prévu: ils s'adaptent automatiquement à l'agencement des murs. Pas besoin de redessiner chaque niveau après une modification, car une fonction maison, ailleurs dans le code, vérifie la position des murs entre eux et les adapte visuellement, à chaque fois que vous passez d'un puzzle à l'autre.

Cependant, vous aurez besoin de modifier le nombre de déplacements maximum que le héros peut faire avant d'être impitoyablement foudroyé. Un bon niveau est généralement conçu pour que la victoire se joue à un déplacement près. Il est aussi nécessaire de préciser l'emplacement exact du héros en début d'épreuve, sinon vous aurez droit à des bugs. Allez donc dans l'onglet principal « DemonGirls » du code et trouvez la fonction : void changeLevel()

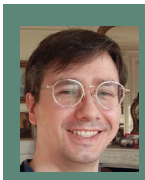
Dans le « switch (levelNumber) » vous voyez que chaque « case » correspond à un niveau. Adaptez les variables rowPlayer et columnPlayer, si besoin. Changez le nombre de coups autorisés « remainingMoves » pour coller à votre nouveau design. La dernière variable « heroTurnsRight » est optionnelle. Elle sert à orienter le héros vers la gauche ou vers la droite en début de niveau. **Figure 18**

Les neuf premiers niveaux de « Demon Girls » sont identiques à ceux de « Helltaker ». Mais le numéro dix est inédit! Il reprend certaines mécaniques de gameplay du reste du jeu, en ajoute une discrètement, et il est en forme de « X » (comme 10 en romain). Comme le dit Keanu Reeves dans « Matrix Awakens »: « On peut aussi se demander pourquoi on choisit une chose au lieu d'une autre. Pourquoi on veut faire X plutôt que Y. »

Alors laissez votre créativité s'exprimer et faites des niveaux aux formes qui vous plaisent. Que vous soyez ange ou démon, Arduboy ou Gamebuino, il y aura toujours une petite place de disponible pour vous dans la communauté des gens qui bricolent des trucs cools.

Idle Factorio : un jeu à barre de chargement 100% Blazor

En tant que framework web, Blazor peut également servir à faire des jeux, particulièrement des jeux de type « idle game ». Quels sont les avantages et inconvénients de Blazor pour faire ce genre de jeux ? Quelles sont les limites de Blazor ? Comment contourner les limites de Blazor ? Cet article présente « Idle Factorio », un jeu vidéo complet, réalisé à 100% en Blazor sans JavaScript, en expliquant de façon logique ses mécaniques de jeu, et en détaillant les principaux points de son architecture technique. Cet article est destiné aux programmeurs souhaitant en apprendre plus sur la conception et la programmation de jeux, tout comme les développeurs voulant en apprendre plus sur Blazor.



Marc Kruzik

Développeur de jeux vidéo depuis 2004, Marc adapte les technologies et techniques du jeu vidéo dans de nombreuses industries (comme la finance, le pétrole, ou la santé), pour améliorer les performances et la prise en main des logiciels informatiques.

http://www.marckruzik.fr

Code source :
https://github.com/marckruzik/Idle_Factorio

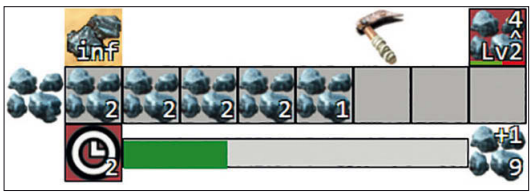
Jouer au jeu en navigateur :
https://idlefactorio.azurewebsites.net

Le projet est inspiré du jeu vidéo d'automatisation d'usines « Factorio » (www.factorio.com), et se concentre plus particulièrement sur sa gestion des ressources.

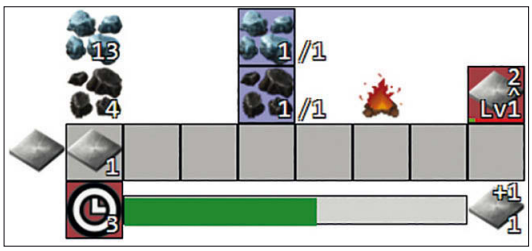
Ce nouveau jeu vidéo nommé « Idle Factorio » est un « idle game actif », c'est-à-dire un jeu où on clique sur des barres de chargement pour gagner des ressources, puis on dépense les ressources pour créer des machines qui automatisent la récolte des ressources. Le nombre d'actions à réaliser et les choix stratégiques à opérer en font un jeu avec de nombreuses actions à mener, d'où l'aspect « actif ». Typiquement, les idle games sont réalisés avec des technologies web, et ont très peu de graphismes. Blazor est donc un excellent candidat pour tester la création de ce genre de jeux. Ici, le projet est réalisé à 100% en Blazor (donc sans JS ni CSS), sous Visual Studio 2019.

Mécaniques de jeu

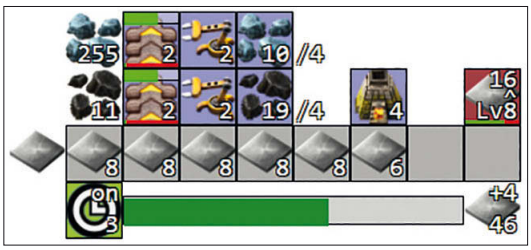
Au début du jeu, on récolte à la main du charbon et du minerai de fer.



On fait fondre le minerai de fer sur un feu de camp pour obtenir des plaques de fer.



Par la suite, on automatise la récolte avec une mine, et on automatise la combinaison avec un four alimenté par des tapis roulants.



Une fois ces aspects de base automatisés, on doit faire des choix critiques sur l'utilisation des ressources : faut-il créer plus de fours ? Ou faut-il d'abord créer plus de tapis roulants ? Ou peut-être plus de mines ?

Les différentes stratégies de jeu conduisent à différentes stratégies de gestion des ressources, qu'il faut équilibrer tout au long de la partie.

Le jeu est extrêmement systémique, car il s'agit d'un jeu de combinaison d'objets. C'est-à-dire que les objets créés servent eux-mêmes de ressources et d'outils pour créer d'autres objets. De cela découle le choix stratégique : est-ce qu'on veut se servir des objets comme des ressources pour obtenir rapidement de nouveaux objets, ou est-ce qu'on veut se servir des objets pour créer de nouveaux outils et multiplier le nombre des futurs objets fabriqués ?

Le jeu propose ainsi un équilibre entre une stratégie court terme (obtenir rapidement des ressources) et une stratégie long terme (investir dans des outils de production)

Phases de jeu

Durant la première phase du jeu, la phase « à la main », tout se fait à la main, action par action :

- La récolte de ressources à la main récolte les ressources et les stocke.
- La combinaison de ressources à la main permet de combiner des ressources existantes en de nouvelles ressources.

Ensuite, la phase d'automatisation permet de créer des outils de production :

- Les récolteurs automatiques de ressources sont les mines.
- Les combineurs automatiques de ressources sont les fours et les machines d'assemblage.

Ce cycle de récolte et d'automatisation se répète ressource

par ressource, au fur et à mesure que le joueur délaisse la gestion des ressources de base pour se concentrer sur des ressources plus évoluées. En début de partie, les ressources de base sont les plus précieuses et deviennent les plus nombreuses. En fin de partie, les ressources finales deviennent les plus précieuses et sont les moins nombreuses.

Mécanismes d'évolution

Le jeu propose des mécaniques d'évolution sur les trois principaux constituants d'un espace de stockage, à savoir :

- L'augmentation de l'espace de stockage se fait avec les coffres.
- L'augmentation de la quantité des transferts se fait avec les bras de chargement.
- L'augmentation de la vitesse des transferts se fait avec les tapis roulants.

Il y a un coffre par ressource, qui dispose d'un nombre de cases fixes. L'espace de stockage des cases peut être augmenté lorsque le coffre est rempli au maximum. C'est une action gratuite à réaliser à la main (non automatisée), qui permet au joueur de « gagner un niveau » sur cette ressource, et qui lui fait prendre conscience du chemin parcouru. En partant de 1 ressource par case, le joueur peut monter jusqu'à plusieurs milliers de ressources par case !

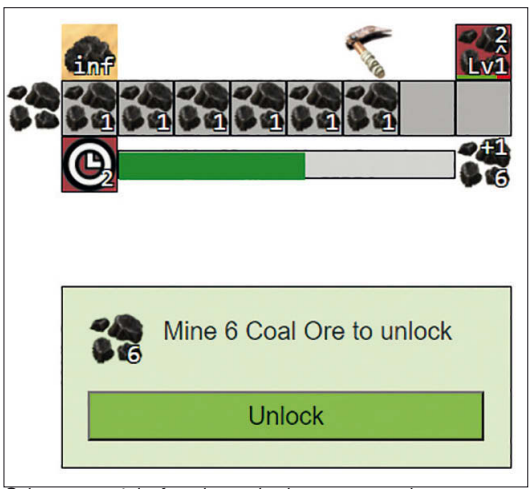
Les tapis roulants sont coûteux en ressources, mais permettent d'accélérer la vitesse de transfert automatique des ressources. Chaque tapis roulant ajouté augmente la vitesse du transfert !

Les bras de chargement sont encore plus coûteux en ressources, mais permettent d'augmenter le nombre de ressources transférées, aussi bien à la main qu'automatiquement. Chaque bras de chargement ajouté transfère une ressource supplémentaire à chaque transfert !

Le système de quêtes

Pour guider le joueur, l'interface de départ du jeu est minimale, et se complexifie en cours de partie.

Un système de quêtes permet de débloquer peu à peu les nouveaux éléments de l'interface. Par exemple, en récoltant suffisamment de charbon, le joueur débloque l'interface du feu de camp, qui permet de consommer le charbon.

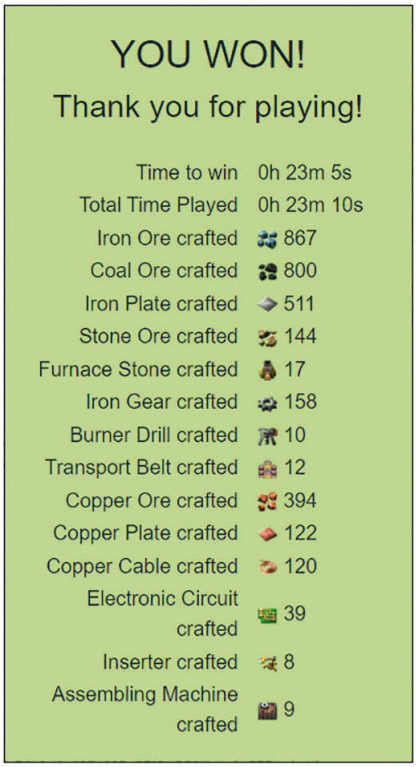


Cela permet à la fois de guider le joueur sur des actions essentielles à maîtriser, et de comprendre le lien logique entre ces actions et les nouvelles parties d'interface dévoilées.

Panneau de victoire

Le jeu se termine lorsque la dernière quête a été accomplie, en ayant fabriqué un certain nombre des ressources de haut niveau, les machines d'assemblage. Là aussi, le joueur doit choisir sa stratégie, entre utiliser les machines d'assemblages pour accélérer sa production, ou les conserver pour atteindre la fin du jeu.

Le panneau de victoire félicite le joueur en lui disant combien de temps il lui a fallu pour terminer la partie, ce qui l'incite à recommencer la partie pour atteindre la fin plus vite. De plus, le panneau de victoire affiche le total de toutes les ressources produites durant la partie, ce qui constitue en quelque sorte le score du joueur. Cela fait prendre conscience au joueur qu'il a commencé en récoltant les ressources une par une, et qu'il termine en automatisant la récolte de dizaines de milliers de ressources. De plus, le score est interactif et continue de mettre à jour les ressources, et le joueur à la satisfaction de voir qu'il a créé un système bien rodé.



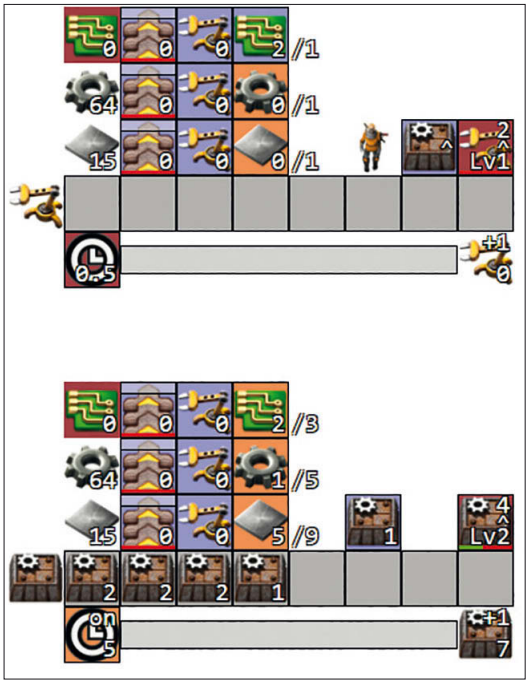
Ce panneau de victoire est une victoire de l'automatisation, à la fois pour la quantité de ressources impliquées qui augmente sans cesse, mais aussi pour l'aspect automatique de l'intégralité du système. En effet, même sans aucune action du joueur, les ressources continuent de s'accumuler, de façon totalement automatisée.

Quadrillage de conception

Dans un environnement graphique, il est beaucoup plus facile de travailler avec des éléments de taille standardisée. Typiquement, les jeux vidéo à l'ancienne utilisent un quadrillage avec des cases de 32 x 32 pixels.

Pour ce projet, les images, extraites du jeu original Factorio, sont de 48 x 48 pixels, et les cases contenant ces images peuvent être de dimension 48 x 48 pixels ou être réglées pour être plus grandes, de façon à avoir une marge supplémentaire.

De façon naturelle, tous les éléments d'interface sont positionnés selon ce quadrillage de 48 x 48 pixels, ce qui accélère considérablement le temps de conception. En effet, lorsque les éléments d'interface sont positionnés sur un quadrillage, on peut facilement les échanger de place, et faire différents essais pour voir quelle est l'interface la plus efficace.
Tout élément d'interface plus grand qu'une case est positionné sur plusieurs cases, et tout élément plus petit qu'une case prend quand même une case entière. Cela permet de se concentrer sur le placement des éléments d'interface au sens large, sans perdre du temps sur des alignements de quelques pixels.



De plus, avoir une case par bouton facilite le développement de l'interface mobile, car ces dimensions de case (environ 1 cm à l'écran) correspondent environ à la taille optimale que doit faire un bouton dans une interface touch.

Cascade contre comportement interne

Blazor promet de faciliter la mise à jour des éléments d'interface, en détectant automatiquement les changements des variables impliquées dans l'interface. Néanmoins, Blazor a besoin que ces variables soient déclarées au niveau de la page, et transmises « en cascade » aux différents composants, pour pouvoir détecter les changements et mettre à jour les éléments d'interface.
Cela pose un problème, car les composants d'interface ont souvent besoin de déterminer eux-mêmes quand et comment se mettre à jour.
C'est particulièrement vrai dans le cas des jeux vidéo, où les comportements sont généralement décidés au niveau des objets eux-mêmes. Plus un objet graphique a un comportement complexe, et plus il a besoin de gérer lui-même la logique de son comportement et son besoin de rafraîchissement graphique.
Créer un projet complexe en Blazor nécessite de mettre en place son propre système de détection de changements et de mise à jour graphique… ce qui revient à ne pas pouvoir profiter d'une partie des avantages promis par Blazor. Blazor permet néanmoins de déclencher une mise à jour graphique avec la méthode StateHasChanged.

Les objets graphiques d'interface : les Game Component

Tous les éléments d'interface héritent de la classe Game_Component, soit en tout 16 classes.
Le positionnement de l'élément d'interface est calculé dynamiquement et renvoyé au format css, ce qui permet un positionnement précis avec des coordonnées au pixel :

```
public string from_position_get_style ()
{
  return $"position: absolute;" +
      $"{(position != null ? $"left: {position.x}px" : "")};" +
      $"{(position != null ? $"top: {position.y}px" : "")};" +
      $"{(position != null ? $"width: {position.width}px" : "")};" +
      $"{(position != null ? $"height: {position.height}px" : "")};";
}
```

Les aspects graphiques les plus simples sont gérés directement dans la partie razor par des tests if, qui décident de l'écriture de l'élément Html. Exemple avec une bordure :

```
@if (border == true)
{
  <div class="layer" style="outline: 1px solid black; outline-offset: -1px;" />
}
```

Les informations textuelles comme les chiffres peuvent apparaître à différents endroits sur les composants (en haut à gauche, en bas à droite, etc.), et sont gérés sous forme de layers, affichés les uns par-dessus les autres. Exemple avec un texte affiché au milieu à droite, avec un positionnement par css calculé en fonction de la hauteur du composant :

```
@if (text_middle_right != "")
{
  <div class="layer">
    <div class="text_wb" style="
    @(position != null ? $"top: {(int)(position.height*0.4)}px;" : "")
    right: 2px;
  ">
      @text_middle_right
    </div>
  </div>
}
```

Le champ list_child contient la liste des enfants, et est automatiquement mis à jour lors de la création des enfants du Game Component. En effet, Blazor ne propose pas de moyen simple permettant à un parent d'accéder à ses enfants, il faut donc que les développeurs ajoutent leur propre méthode.

Évolution d'interface

Comme le jeu est avant tout un jeu à combinaison de ressources, la plupart des éléments de jeu se répètent, tout comme les interfaces. En effet, chaque ressource a une interface relativement similaire à celle des autres ressources, et lorsqu'un stade supplémentaire est atteint (comme un changement d'espace de stockage du coffre ou quand des outils

d'automatisation sont ajoutés), l'interface évolue et se complexifie. Ici, Blazor remplit bien son rôle de mise à jour des éléments d'interface, et c'est un vrai point fort de la technologie.

Base de données CSV

Les données du jeu sont conservées au format CSV, ce qui permet de les éditer sous forme de tableaux à double entrée. À ce stade, le JSON n'est pas assez normalisé, et une base de données est trop restrictive. En effet, les différents éléments du jeu sont normalisés, avec un certain nombre d'éléments, ce qui rend leur ajout et leur équilibrage plus facile à réaliser dans un tableau.

Attention, par défaut, HttpClient utilise le cache du navigateur pour stocker les CSV. Quand on souhaite pouvoir modifier souvent les CSV, il faut éviter de mettre les CSV en cache, et donc préciser dans les Headers que le CacheControl doit avoir NoCache à true.

```
HttpClient http = new HttpClient ();
http.BaseAddress = new Uri (Program.base_adress_str);
http.DefaultRequestHeaders.CacheControl = new CacheControlHeaderValue
{
  NoCache = true
};
Stream st = await http.GetStreamAsync (csv_filepath);
```

Amélioration possible : une fois la phase de design vraiment terminée, les données du jeu pourraient être stockées dans une base de données reprenant les fichiers et colonnes CSV sous forme de tables.

Langage de recettes

L'intégralité du jeu consiste à fabriquer des objets, qui sont par la suite utilisés dans d'autres recettes. La création d'un « langage de recettes » facilement interprétable permet de gérer les 14 objets du jeu. Utiliser 1 fer (iron) et 1 charbon (coal) permet de fabriquer 1 plaque de fer (iron plate) et s'écrit sous cette forme :

```
iron_ore * 1 + coal_ore * 1 = iron_plate * 1
```

En fonction du lieu de fabrication (feu de camp, four, machine d'assemblage), l'interface change légèrement, mais le principe reste le même : chaque recette est dynamiquement chargée dans la page sous forme d'interface pour réaliser la recette.

Une recette avec 1 ingrédient a 1 ligne en entrée, une recette avec 3 ingrédients a 3 lignes en entrée, etc. Cette génération dynamique d'interface autour du langage de recette permet de traiter facilement les 14 objets du jeu, et d'étendre potentiellement le jeu à des dizaines d'autres recettes.

Horloge logique et graphique

Dans les moteurs de jeux vidéo, la logique (comme la physique) tourne généralement à 60 itérations par secondes, les graphismes tournent à 60 itérations par secondes (FPS) ou plus. Les technologies web ne sont pas faites pour des mises à jour à 60 itérations par seconde, mais il est possible d'obtenir un comportement proche.

Un Timer peut lancer une fonction à intervalles à peu près réguliers et ainsi simuler une logique à 60 itérations par seconde. Il convient toutefois de contrôler le temps réel écoulé avec un StopWatch, de façon à s'assurer que le temps logique soit bien respecté. Grâce à cela, en cas de lag qui provoque un retard de déclenchement du Timer, la logique du jeu tourne en boucle jusqu'à rattraper le temps réel écoulé et donne l'impression que le temps s'écoule de façon stable quoi qu'il arrive.

```
public void StartTimer ()
{
  if (this.timer_logic == null)
  {
    this.timer_logic = new System.Timers.Timer (this.logical_interval);
    this.timer_logic.Elapsed += NotifyTimerElapsed;
    this.timer_logic.AutoReset = true;
    this.timer_logic.Enabled = true;
  }
  this.stopwatch = new Stopwatch ();
  this.stopwatch.Start ();
  this.logical_time_elapsed = 0;
}


private void NotifyTimerElapsed (object source, ElapsedEventArgs e)
{
  while (this.logical_time_elapsed < this.stopwatch.ElapsedMilliseconds)
  {
    logical_update ();
    this.logical_time_elapsed += this.logical_interval;
  }
}
```

Pour l'horloge graphique, il faut une mise à jour extrêmement rapide, et une prise en compte du lag. Après de nombreux essais, les meilleures performances sont obtenues avec une boucle infinie asynchrone (pas de Timer) et un simple Delay.

```
async Task graphical_thread_task ()
{
  while (true)
  {
    if (this.graphical_running == false)
    {
      break;
    }
    graphical_update ();
    await Task.Delay (this.graphical_interval);
  }
}
```

Fausse bonne idée : faire un calcul dynamique de l'intervalle graphique, sur plusieurs tours d'horloge graphique, pour rallonger le temps de Delay en cas de lag et laisser plus de ressources à la machine. C'est une fausse bonne idée, car les lags sont extrêmement courts, et il vaut mieux toujours mettre à jour graphiquement le projet quand on peut. C'est la régularité de l'horloge logique qui donne l'impression que le projet reste fluide.

Calcul de conditions

Les conditions de jeu dépendent souvent de paramètres multiples, qui sont vérifiés à chaque tour d'horloge logique. Exemple pour une recette réalisée à 50 % contenant du charbon :

```
recipe.percentage > 50 && recipe.contains ("coal")
```

Néanmoins, réaliser des calculs de conditions à chaque tour d'horloge logique se révèle être un gouffre de performances pour une technologie web. Cette approche a donc été évitée le plus possible au profit d'observables, qui modifient l'interface uniquement si certaines données ont changé.

Observables

Dans les moteurs de jeux, en plus de la logique et des graphismes, l'interface (les fenêtres et boutons) peut être rafraîchie plusieurs centaines de fois par seconde. En effet, le moteur de jeu redessine entièrement l'interface, car les quelques centaines d'éléments d'interface représentent très peu de chose à côté des centaines de milliers (ou millions) de polygones pour les décors et personnages.

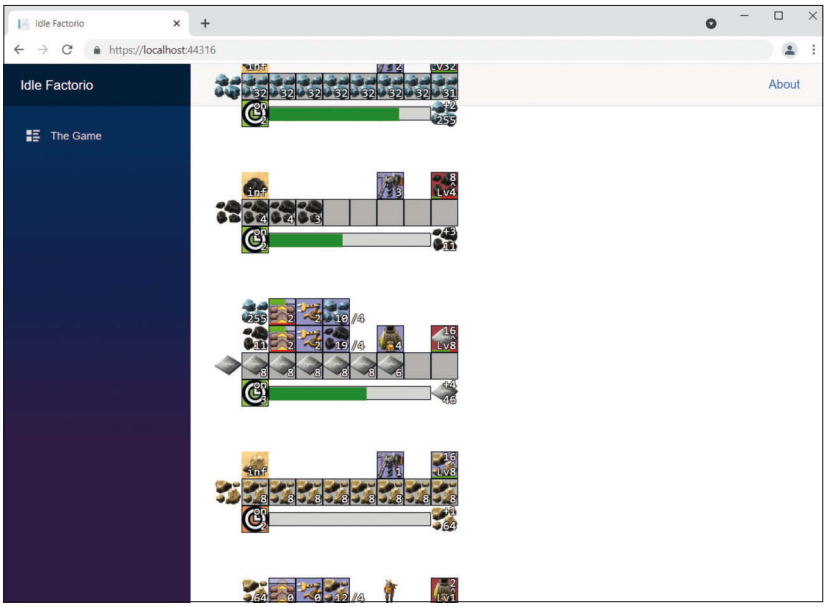
Dans une interface web, il est hors de question de redessiner l'interface depuis zéro. De plus, il faut s'assurer que chaque partie de l'interface soit rafraîchie uniquement si besoin, voire de contrôler les sous-parties de l'interface qui se rafraîchissent sans toucher à leurs parents.

Le projet propose donc un système d'observable simple, et les composants d'interface du projet peuvent se brancher sur l'observable d'une ressource en particulier. Par exemple, la modification de la quantité globale de charbon affecte l'interface de création de charbon, mais aussi chaque recette utilisant du charbon, en modifiant le moins d'éléments possible. De plus, toute modification d'élément ne rafraîchit pas directement l'élément, mais passe un flag need_refresh à true, de façon à ce que des modifications multiples attendent le prochain tour d'horloge graphique pour être véritablement rafraîchis.

Amélioration possible : utiliser une librairie d'observables (comme RX.Net), et mettre en place un système de temporisation des rafraîchissements pour espacer ceux-ci dans le temps quand ils sont très fréquents.

Améliorations potentielles

- Le projet dépasse souvent les 300 mises à jour d'éléments graphiques par seconde, et la très grande majorité sont les mises à jour des barres de chargement. Bien que le but du projet soit d'être à 100% en Blazor, convertir le remplissage des barres de chargement en animations CSS donnerait un gain de performance considérable.
- L'interface est plus longue que l'écran, particulièrement sur mobile. Éviter d'animer les éléments en dehors de l'écran donnerait un gain de performance considérable.
- Un système de sauvegarde automatique (et de remise à zéro) permettrait aux joueurs de fermer l'onglet et d'y revenir. Car sans sauvegarde, il faut faire la partie en une fois sans fermer ou recharger la page.



Conclusion

Ce jeu a permis de pousser Blazor dans ses retranchements et de se rendre compte des limitations en termes de performances de rafraîchissement, qui sont tout de même de plusieurs centaines de rafraîchissements par seconde. Pour du jeu vidéo, c'est peu, mais pour du web, c'est beaucoup.

Le garbage collector semble fonctionner correctement, et aucune fuite mémoire n'a été observée.

Pour s'adapter aux besoins du jeu vidéo, qui ont de nombreux éléments graphiques dynamiques, Blazor a plusieurs pistes d'amélioration :

- Blazor pourrait proposer plus de possibilités pour les éléments à se rafraîchir par eux-mêmes, plutôt que de tout miser sur son système de cascade.
- Blazor pourrait s'améliorer sur les performances de rafraîchissement, en permettant à des éléments d'interface précis de se mettre à jour plusieurs dizaines de fois par seconde.
- Blazor pourrait économiser sur les performances de rafraîchissement, en consacrant moins de ressources aux éléments d'interface en dehors de l'écran.
- Blazor pourrait proposer une horloge régulière précise pouvant monter à 60 itérations par seconde, c'est-à-dire une fonction d'update logique lancée régulièrement toutes les 16 millisecondes.

D'un point de vue extérieur, les technologies web sont souvent statiques, héritées du besoin lié au rendu HTML. JavaScript a longtemps été le passage obligatoire pour accéder au dynamisme d'un rafraîchissement partiel des pages internet. Blazor nous promet de pouvoir réaliser des cas complexes sans JavaScript, et ce projet en est la preuve. Certes, JavaScript peut être utilisé en plus de Blazor pour améliorer de nombreux cas d'usage, mais la promesse de pouvoir utiliser Blazor sans JavaScript est tenue.

En conclusion, Blazor améliore le dynamisme des technologies web, et c'est un pas dans la bonne direction.

À vous de coder !

Développer des jeux 8-bit avec Cross-Lib

Cross-Lib est un projet dont la finalité est le retro-développement (retro-coding). Celui-ci est lié au retro-computing et au retro-gaming. Tous ces termes sont liés aux anciens ordinateurs et consoles, mais il faut d'abord clarifier leur définition.

Fabrizio Caruso
fabrizio_caruso@hotmail.com
https://github.com/Fabrizio-Caruso/CROSS-LIB

Fabrizio, sicilien, mais niçois d'adoption depuis 9 ans, ingénieur en informatique, est passionné par la programmation pour d'anciens ordinateurs et consoles et par l'histoire de l'informatique, sur lesquelles il a travaillé dans le cadre de plusieurs projets. Ses travaux sont concentrés sur la programmation en C pour les machines 8-bit des années 80 ; sur la programmation en BASIC des années 80 et son histoire. Il participe régulièrement à des compétitions de programmation pour les ordinateurs 8-bit en C ou BASIC avec plusieurs jeux dans la compétition internationale BASIC 10-liner (classé deuxième en 2021).

Qu'est-ce que le *retro-gaming*, le *retro-computing* et le *retro-développement* ?

- Le *retro-gaming* est probablement le plus connu des trois. Il s'agit de l'amour pour les jeux sur d'anciens ordinateurs et consoles.
- Le *retro-computing* est le hobby plus générique qui consiste à utiliser des ordinateurs et consoles anciens pas seulement pour y jouer, mais aussi pour d'autres activités telles que le traitement de texte.
- Le retro-développement (ou en anglais *retro-coding*) est le développement de logiciels pour d'anciennes machines, c'est-à-dire ordinateurs, consoles, bornes d'arcades, calculatrices scientifiques, etc.

Cross-Lib est un outil de *retro-coding* libre et open source pour plus de 200 machines 8-bit des années 80 (ordinateurs, consoles, calculatrices, bornes d'arcade, etc.) comme le *Commodore 64*, le *Commodore VIC 20* et les autres *Commodores*, les *Thomsons Mo5* e *To7*, l'*Apple][*, l'*Atari 800*, l'*Amstrad CPC*, le *Sinclair ZX 81 et ZX Spectrum*, les *MSX*, l'*Oric 1* et *Atmos*, le *BBC Micro*, le *TRS-80 CoCo*, le *Dragon 32*, la *GameBoy*, la *Sega Master System*, la *GameGear*, la *Nintendo NES*, les calculatrices Texas Instruments comme la *TI 83*, etc. avec pour principales architectures Zilog 80, MOS 6502 et Motorola 6809. Une liste partielle avec plus de 160 machines supportées se trouve à la page : https://github.com/Fabrizio-Caruso/CROSS-LIB/blob/master/docs/STATUS.md. Cross-Lib supporte des machines plutôt rares comme le *Jupiter Ace*, etc., et des machines très exotiques par exemple plusieurs ordinateurs de l'Europe de l'Est comme le yougoslave *Galaksija*, le hongrois *Homlab-2*, plusieurs ordinateurs de la série *Robotron* comme le *Robotron KC 85*, etc. Cela pourra sûrement intéresser aussi les passionnés de *retro-computing* et de *retro-gaming* parce que pouvoir coder sur les retro-ordinateurs et consoles permet de produire des outils et des jeux pour ces machines.

MOTIVATION : Pourquoi fait-on du *retro-coding* ?
Il constitue une source de plaisir, une façon de s'entraîner à coder pour des machines avec des ressources très limitées [c8b] comme on ferait pour des microcontrôleurs [effc], de la nostalgie et un moyen de faire des « découvertes historiques ou archéologiques » (par exemple le travail académique [east] fait par Caruso *et al.*).

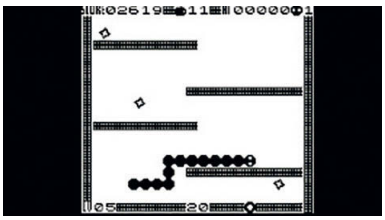
DISCLAIMER : Il faut préciser que Cross-Lib dans son état actuel n'est pas complet, mais est déjà utilisable. Ici, je présente Cross-Lib dans son état de développement actuel.

Certains scripts de Cross-Lib peuvent effacer des fichiers. L'auteur décline toute responsabilité pour la perte de fichiers.

1. RÉSULTATS POSSIBLES

Ici figurent des captures d'écran des cinq jeux que j'ai écrit avec Cross-Lib (voir [xlib], [app], [ceo]). Les images montrent les résultats sur des machines différentes où, par exemple, le nombre de couleurs disponibles diffère. Les graphismes possibles avec Cross-Lib seront ceux que l'on pouvait trouver dans les ordinateurs du début des années 80, par exemple sur le *Sinclair ZX Spectrum*.

CROSS SNAKE
(Sinclair ZX Spectrum, GameBoy, Commodore VIC 20)



CROSS HORDE
(Sega Master System, Amstrad CPC, Atari 800)

CROSS BOMBER (Atari 800), CROSS CHASE (Atari 800), CROSS SHOOT (Sega Master System)

2. RETRO-CODING SANS CROSS-LIB

Aujourd'hui, il y a beaucoup d'outils qui permettent de coder pour les retro-machines très facilement sur un ordinateur moderne. Dans ce cas-là on parle de cross-développement.
Historiquement, les machines 8-bit étaient programmées en Assembleur ou, pour des programmes plus simples, en BASIC interprété.
Le cross-développement simplifie beaucoup la tâche du développeur (éditeurs modernes, debugging simplifié, compilation efficace et rapide, etc.). La programmation pour ces machines se fait encore très souvent en Assembleur pour pouvoir exploiter au maximum les ressources hardware de chaque machine, mais on ne disposera pas de code portable.
En revanche, grâce à des cross-compilateurs et bibliothèques très performantes, il existe des alternatives comme le C, qui est un bon compromis entre performance et rapidité d'écriture du code et permet d'avoir du code au moins partiellement portable.

La plupart des retro-développeurs veulent exploiter une machine particulière au maximum. Dans le passé, la portabilité n'a pas été une priorité pour le retro-coding parce qu'elle est difficile à mettre en pratique sur des machines si différentes et parce qu'elle impose des limitations sur la possibilité d'exploiter toutes les caractéristiques de chaque machine. L'exemple le plus évident de retro-développement qui exploite les limites (et en effet dépasse les limites hardware connues) sont les productions de la *demo-scene*. On voit des jeux très récents pour certains ordinateurs 8-bit (comme les Commodore, les Sinclair et les Atari) qui ont des graphismes que l'on a cru impossibles sur ces machines jusqu'il y a 20 ans, bien que rien n'ait été changé dans le hardware. Par exemple il y a des jeux avec sprites multi-couleurs pour le Sinclair ZX Spectrum qui n'a ni sprites hardware ni modes graphiques multi-couleurs.

Récemment on a vu apparaître des outils qui permettent la production de code partiellement portable ou des outils pour plusieurs machines différentes. Un exemple est le dev-kit *Z88DK* avec ses bibliothèques pour le développement en C sur presque une centaine d'ordinateurs et consoles basés sur les architectures Zilog 80 et Intel 8080. Deux autres exemples sont le dev-kit *CC65* et le dev-kit *8-Bit-Unity* [un8] (basé sur *CC65*) pour le développement en C sur certaines architectures basées sur le MOS 6502. Grâce à ces outils on a commencé à voir des jeux 8-bit multi-plates-formes qui n'exploitent pas forcement tous les détails de chaque machine, mais qui sont très soignés par exemple *8-bit Slicks* écrit avec *8-Bit-Unity* et *CC65*.

Le problème de la portabilité du code C est lié au fait que les machines sont très différentes et que le langage C n'a pas de bibliothèques standard pour les graphismes, les sons, l'input de manettes, etc. Une solution partielle est donnée par certains dev-kits comme *CC65*, *Z88DK*, *LCC1802* (pour l'architecture RCA 1802), *CMOC* (pour l'architecture Motorola 6809), etc. En utilisant l'un de ces dev-kits, on peut utiliser des bibliothèques pour l'input et l'output qui sont compatibles sur plusieurs machines de la même architecture.

En revanche deux problèmes se posent :
1. Chaque dev-kit ne couvre qu'un sous-ensemble des architectures 8-bit.
2. Les bibliothèques fournies avec les dev-kit ne couvrent pas toujours tous les graphismes, le son ou l'input sur toutes les machines supportées.

3. AVANTAGES ET LIMITES DE CROSS-LIB

Avec Cross-Lib, on veut faire plus que ce qui est possible avec les autres outils :
1. couvrir toutes les architectures 8-bit, y compris les plus exotiques ;

2. permettre l'écriture de code 100% *WORA* ("*write once, run anywhere*") ;
3. permettre la création de ressources graphiques *WORA*, c'est-à-dire que l'on pourra définir les éléments graphiques (les « tiles ») une seule fois pour (presque) toutes les machines ;
4. fournir une *tool-chain* simplifiée pour créer, compiler et tester les jeux.

J'ai choisi le nom *Cross-Lib* comme contraction de "*cross-compilable library*" (bibliothèque logicielle cross-compilable). Donc Cross-Lib est (surtout) une bibliothèque logicielle pour écrire du code en C qui peut être compilé par plusieurs cross-compilateurs pour presque toutes les consoles et ordinateurs 8-bit. On pourrait dire que le but principal de Cross-Lib est de fournir une abstraction qui puisse couvrir toutes les machines 8-bit. Il permet l'écriture de jeux universels pour plus de 200 machines différentes.

Cross-Lib n'est pas un compilateur. Il a besoin d'un, voire plusieurs compilateurs pour pouvoir produire des fichiers binaires (exécutables, images des disquettes, images des cassettes, images des cartouches, etc.) pour les retro-ordinateurs et retro-consoles.

Cross-Lib est aussi un ensemble d'outils (scripts) qui permettent la compilation universelle et l'adaptation automatique des ressources graphiques (« graphics assets ») à toutes les machines.

Le fait que Cross-Lib fournisse une couche d'abstraction implique beaucoup de compromis par rapport à ce que l'on pourrait faire avec une machine spécifique si l'on ne voulait programmer que spécifiquement pour cette seule machine. Notamment Cross-Lib permet de programmer des graphismes limités parce qu'on veut pouvoir les afficher sur toutes les machines équipées de graphismes. On aura également des effets sonores, mais très limités.

Il y a de nombreux dev-kits pour programmer pour plusieurs machines, mais très peu permettent de le faire de façon *WORA* par exemple *Z88DK* et *8-Bit-Unity*. Il y a en d'autres qui ne sont pas *WORA* et qui, dans certains cas, n'utilisent pas forcément le langage C comme c'est le cas de *TRSE* [trse] (« *Turbo Rascal Syntax Error* »). Ici on fait une comparaison entre des dev-kit *WORA* :

	Inclut compilateur(s)	Ressources WORA	API pour les sprites	Nombre de machines	Architectures
Cross-Lib	Non	Oui	Non	~200	Intel 8080, MOS 6502, Motorola 6809, RCA 1802 et autres
8-bit Unity	Non	Non	Oui	5	MOS 6502
Z88DK	Oui	Oui	partiellement	~100	Intel 8080, Zilog 80

Ce tableau montre les points forts et faibles de Cross-Lib : il est le seul dev-kit WORA qui permet de coder sur (presque) toutes les architectures et machines 8-bit. Par contre Cross-Lib ne propose pas d'API pour les *sprite* et en général pas d'API avancées. La raison est que Cross-Lib couvre toutes les machines, y compris celles qui n'ont pas de graphismes ou pas de *sprites* hardware ou pas de modes graphiques qui permettent d'implémenter des *sprites* software génériques. La seule manière d'implémenter des *sprites* avec Cross-Lib est l'utilisation de *pre-shifted tiles* dont on parlera dans les sections suivantes.

4. COMPILATEURS C ET LEURS LIMITES

Cross-Lib est censé être utilisé avec des compilateurs C, mais lesquels ?

Un compilateur natif (par exemple GCC) peut être utilisé pour produire des fichiers exécutables natifs qui tournent par exemple sous Windows. Aujourd'hui le support pour la compilation native est limité au terminal. Donc pas de graphismes sur la machine native.

Actuellement Cross-lib supporte pleinement les cross-compilateurs et dev-kits avec graphismes et effets sonores :

- **CC65** [cc65] pour la plupart des machines basées sur l'architecture MOS 6502 ;
- **Z88DK** [z88dk] pour les machines basées sur Zilog 80 et Intel 8080 ;
- **CMOC** [cmoc] pour les machines basées sur Motorola 6809 ;
- **LCC1802** [lcc1802] pour les machines basées sur le COSMAC RCA 1802.

Il supporte aussi d'autres compilateurs (pour l'instant sans graphismes et sans effets sonores) comme ACK [ack] for PC 8088/8086, CP/M Intel 8080, 386/68K/PPC/MIPS, PDP11, XTC68 [xtc68] pour le Sinclair QL, VBCC [vbcc] pour le BBC Micro, BBC Master, l'Amiga et d'autres, CC6303 [cc6303] pour le Motorola 6803, et plusieurs versions de GCC modifiées pour cibler des retro-ordinateurs (Atari ST [st], Texas Instruments TI99/A [ti], Olivetti M20 [m20]).

Pourquoi utilise-t-on un sous-ensemble du standard C89 ?

Parce que la plupart des cross-compilateurs disponibles pour les architectures 8-bit n'implémentent qu'un sous-ensemble du C89 avec très peu de C99 (les commentaires //).

Donc on écrit en C (un sous-ensemble du ANSI C C89) avec les API de Cross-Lib pour les graphismes, les effets sonores, l'input (clavier/joystick/manette) et d'autres fonctions qui permettent l'écriture de code universel quand l'équivalent du standard C n'est pas défini de manière univoque.

En particulier il faut programmer en C89 sans

- "float" et "double",
- copies et passage par valeur d'objets de type "struct",
- le heap,
- fonctions récursives.

La contrainte sur les fonctions récursives réside dans son coût, surtout pour l'architecture MOS 6502. On pourrait quand même utiliser des fonctions récursives si cela était vraiment nécessaire.

5. INSTALLATION

Cross-Lib nécessite :

1. un environnement POSIX (Linux, FreeBSD, Cygwin sous Windows, Windows Subsystem for Linux, etc.) ;

2. des outils de build (GNU make) et de développement (python, java) ;
3. au moins un compilateur ANSI C natif et/ou un des cross-compilateurs supportés ;
4. des émulateurs (optionnels, mais très utiles).

Il n'y a pas une procédure standard pour installer tous les outils et les compilateurs parce que cela dépend de l'environnement spécifique choisi. Ici on donnera des exemples et des conseils génériques pour l'installation. Un exemple d'installation très détaillée se trouve dans l'annexe à la fin de cet article. Pour simplifier, on a prévu pour le futur une version de Cross-Lib dans une image *Docker* avec tous les outils et compilateurs.

5.1 OUTILS

Il suffira d'installer GNU make et python (version 2.x ou 3.x).

Sous Linux il existe des commandes qui simplifient l'installation. Par exemple sur certaines distributions Linux on pourra utiliser apt-get install make.

Sous Cygwin/Windows, cela se fait à travers son installer en choisissant le package make.

Sous FreeBSD il faudra installer gmake parce que make sur les Unix de la famille BSD lance par défaut BSD make, qui n'est pas compatible avec Cross-Lib.

5.2 COMPILATEURS ANSI C

Il faut installer au moins un des compilateurs supportés. Cross-Lib ne supporte que les toutes dernières versions des cross-compilateurs.

Pour commencer, il faudrait installer un compilateur natif lequel nous permettra de compiler des versions natives, c'est-à-dire pour le terminal qui n'a pas des graphismes, mais cela nous permettra de démarrer et voir des résultats. En plus un compilateur natif est utile pour tester et debugger le code. GCC est un compilateur natif qui est facile à installer. Par exemple sur Ubuntu et d'autres distributions Linux qui utilise apt il pourrait suffire :

```
sudo apt install build-essential
```

Installation de Cross-Compilateurs et Dev-kits 8-bit : la majorité des machines supportées est couverte par deux dev-kits:

- Z88DK (qui supporte environ une centaine de machines)
- CC65 (qui supporte plus de 30 machines)

La procédure d'installation de CC65 dépend de l'environnement. En général on peut toujours compiler le code source.

Sous Windows on télécharge la *snapshot* de CC65 (voir [cc65] pour le lien), le décompresser et rajouter la location de cc65-snapshot-win32/bin à la variable PATH. Cela suffira pour utiliser CC65 sous Windows. Sur Linux, FreeBSD ou macOS, on peut, soit compiler la source, soit installer depuis un répertoire (si disponible). Par exemple sur Ubuntu on pourrait installer CC65 avec :

```
sudo apt-get install cc65
```

Si l'on utilise un package précompilé, il faut vérifier que le répertoire contient une version récente.

Avec une version ancienne, on risque d'avoir de problèmes de compatibilité avec Cross-Lib.

Sous Cygwin/Windows, on peut tout simplement installer CC65 sur Windows et le lancer depuis Cygwin.

Comme pour CC65, la procédure d'installation du dev-kit Z88DK dépend de l'environnement.

Une description des différentes procédures est disponible sur :
https://github.com/z88dk/z88dk/wiki/installation

Par exemple sous Linux on peut suivre :
https://github.com/z88dk/z88dk/wiki/installation#linux—unix

Sous Windows, la solution la plus simple est d'utiliser une des snapshots précompilées disponibles sur http://nightly.z88dk.org/ e définir la variable d'environnement ZCCCFG comme décrit dans la procédure d'installation spécifique de l'environnement.

6. PROGRAMMER EN C AVEC CROSS-LIB

Coder avec CROSS-LIB implique trois choses :

1. programmer en C (un sous-ensemble du standard C89) un peu comme on le ferait si l'on codait pour un microcontrôleur. Ne pas oublier les limites des appareils 8-bit des années 80 ;
2. utiliser les API de Cross-Lib pour l'input/output et pour très peu d'autres choses ;
3. programmer de façon abstraite grâce aussi à certaines macros pour tous les machines supportées, y compris les machines qui ont des fonctionnalités réduites.

Le langage C est bien adapté pour des architectures à 16, 32 et 64-bit et beaucoup moins pour les 8-bit des années 80, qui sont très proches des microcontrôleurs 8-bit d'aujourd'hui (voir [c8b] et [effc] pour plus de détails).

Par exemple il faudra:

- éviter les fonctions récursives parce que certaines architectures n'ont pas de stack ou elles ont un stack hardware très peu profond ;
- se limiter aux types unsigned à 8 et 16 bit (uint8_t et uint16_t) parce que les opérations avec signe ne sont pas très efficientes sur les architectures vintage à 8-bit ;
- réduire le nombre des variables locales pour limiter l'utilisation du stack ;
- réduire le nombre des paramètres pour limiter l'utilisation du stack ;
- réduire le nombre d'unités de compilation (nombre de fichiers `.c` avec leurs *headers*) parce que de nombreux compilateurs n'optimisent pas le code qui est partagé entre plusieurs unités de compilation différentes (pas de « *link-time optimization* ») ;
- éviter, lorsque cela est possible, les produits, les divisions et le modulo en utilisant des opérateurs sur les bits (comme `&`, `<<`, `>>`, etc.) parce que le produit et la division sont des opérations très lourdes ;
- parfois optimiser la mémoire plutôt que la performance parce que certaines machines ont une quantité minuscule de mémoire vive (dans certains cas moins d'un kilooctet).

7. COMMENCER AVEC CROSS-LIB

Il faut comprendre la structure des dossiers de Cross-Lib. Tout d'abord il faudra se mettre dans le répertoire src/ où se trouve tout le code C et les scripts Python.

Le code de Cross-Lib se trouve dans le répertoire src/cross-lib/.

On appellera *projet* un logiciel écrit avec Cross-lib. On a 3

types de projets : les jeux built-in, les exemples built-in et les nouveaux projets définis par l'utilisateur.
Le code C des projets se trouve dans :

- src/games/ qui contient le code des jeux built-in ;
- src/examples/ avec plusieurs exemples ;
- src/projects/ qui contient le code des nouveaux projets définis par l'utilisateur et c'est le seul code que l'utilisateur est censé écrire et modifier.

Le script xl est un script Python qui simplifie la création de nouveaux projets et leur compilation, test et exécution.
Ce script doit être utilisé depuis le répertoire src/ et peut effacer des fichiers donc il faudra l'utiliser avec précaution.
On montrera comment utiliser xl avec des exemples. Pour avoir les instructions complètes on peut lancer xl help depuis le répertoire src/.
Cross-Lib est fourni avec de nombreux exemples de code qui montrent comment l'utiliser. Tous les exemples simples se trouvent dans src/examples/. En plus on a le code des 5 jeux complets dans src/games/.
En particulier on peut commencer avec helloworld qui se trouve dans src/examples/helloworld/, dans lequel on trouve le fichier main.c qui contient :

```
#include "cross_lib.h"

int main(void)
{
  _XL_INIT_GRAPHICS();

  _XL_CLEAR_SCREEN();

  _XL_SET_TEXT_COLOR(_XL_WHITE);

  _XL_PRINT_CENTERED("HELLO WORLD");

  while(1){};

  return EXIT_SUCCESS;
}
```

On remarque tout de suite que les fonctions de Cross-Lib commencent par _XL_ :

1. _XL_INIT_GRAPHICS() qui est nécessaire pour initialiser le display ;
2. _XL_CLEAR_SCREEN() qui efface l'écran ;
3. _XL_SET_TEXT_COLOR(_XL_WHITE) qui définit la couleur du texte qui suivra ;
4. _XL_PRINT_CENTERED("HELLO WORLD") qui affiche la chaîne de caractères « HELLO WORLD » centrée au milieu de l'écran.

Pour compiler un projet, on peut utiliser le script xl avec la syntaxe suivante :

```
xl <project> <target>
```

où <project> est le nom du project et <target> est le nom de la machine (par défaut : ncurses, c'est-à-dire la build compilée avec GCC pour le terminal natif avec ncurses).
Donc pour compiler helloworld avec GCC et ncurses on fera :

```
xl helloworld
```

Il produira un fichier exécutable natif dans le répertoire build qu'on pourra lancer avec

```
xl run helloworld
```

Pour cross-compiler helloworld, il faudra spécifier une *target* différente. Par exemple

```
xl helloworld c64
```

On pourra lancer avec un émulateur pour le Commodore 64 et si l'on a installé l'émulateur x64 (émulateur C64 contenu dans Vice), on pourrait le lancer directement depuis le terminal avec

```
xl run helloworld c64
```

Dans la plupart des cas, Cross-Lib produira des images binaires (image descassettes, disquettes, cartouches, etc.) prêtes à être utilisées avec des émulateurs ou avec des machines réelles. Actuellement xl run ne supporte qu'un nombre très limité d'émulateurs. En général il faudra lancer un émulateur et charger l'image produite par Cross-Lib depuis l'émulateur. On trouve plus de détails dans la page *GitHub* du projet [xlib].
Pour charger les images sur des machines réelles, il faudra soit les convertir dans un format lisible pour la machine spécifique soit utiliser un périphérique moderne, par exemple *SD2IEC* pour les Commodores 8-bit et les images de type .prg et .d64, depuis une machine réelle en émulant un lecteur de disquettes. Pour la plupart des images des cassettes, on peut les convertir dans un format lisible par le magnétophone (par exemple en format WAV encodé spécifiquement pour la machine) avec des outils spécifiques pour chaque machine. Il faudra connecter la sortie audio du PC à l'entrée audio du retro-ordinateur. Dans certains cas, on pourra utiliser un smartphone ou lecteur MP3 à la place du PC et le connecter à un adaptateur de cassettes de voiture (celui qu'on utilise normalement pour reproduire des MP3 ou WAV dans un magnétophone) et on mettra l'adaptateur de cassettes dans le magnétophone du retro-ordinateur.

8. LE FLUX DU DÉVELOPPEMENT

Dans ce diagramme on décrit le flux de développement : **figure 1**
On voit les opérations que l'utilisateur peut faire pour coder avec Cross-Lib, notamment :

1. écrire du code abstrait avec les API Cross-Lib ;
2. dessiner les graphismes (*tiles*) avec un outil externe ou directement en éditant les ressources graphiques (« assets ») ;
3. utiliser le script xl pour compiler, lancer, tester, etc.

Le flux de développement complet avec les opérations déclenchées par la *tool-chain* est décrit par le diagramme suivant : **figure 2**

9. CRÉER UN NOUVEAU PROJET

Un projet est juste un dossier dans src/games/, src/examples/ ou src/projects. Ce dernier répertoire contient les projets définis par l'utilisateur.
On pourrait créer manuellement un répertoire avec des

fichiers `.c`, un répertoire tiles (avec tous ses sous-répertoires) et un fichier Makefile.<project>, mais il est possible de faire tout cela plus simplement avec

```
xl create <project> <type>
```

<project> est le nom du nouveau projet et le paramètre optionnel <type> est le type (arcade, text, helloworld, test) de projet qui définira le code initial du projet.
Par exemple :

```
xl create foo arcade
```

On crée un projet foo avec du code initial pour un jeu de type arcade (avec certaines routines pour gérer score, record, input, etc.).
On pourra le compiler pour le terminal natif avec :

```
xl foo
```

Ou cross-compiler pour une machine, comme par exemple la *GameBoy*, avec

```
xl foo gb
```

Une autre manière de créer un projet est de cloner un projet avec :

```
xl clone <source project> <new project>
```

Par exemple, on peut cloner le jeu *Cross Horde* et créer un projet bar qui au début aura le même code avec :

```
xl clone horde bar
```

10. LE DISPLAY

Dans cette section on montrera les principales fonctions pour afficher des objets sur l'écran.

10.1 INITIALISER ET EFFACER L'ÉCRAN

Tout d'abord avant de faire quoi que ce soit il faudra initialiser le display avec

```
void _XL_INIT_DISPLAY(void)
```

Avant de commencer à utiliser l'écran, il faudra aussi effacer le contenu de l'écran avec

```
void _XL_CLEAR_SCREEN(void)
```

comme on a vu dans l'exemple qui affiche « HELLO WORLD ».

10.2 LES COULEURS

Rappelons que chaque machine spécifique peut ne pas avoir de couleurs ou avoir un nombre très limité de couleurs possibles. Quand on code avec Cross-Lib, on choisit des couleurs abstraites qui, ensuite, seront approximées sur chaque machine spécifique de façon différente. Sur les machines qui n'ont pas de couleurs, le choix d'une couleur n'aura aucun effet. Sur les machines où une couleur est absente, cette couleur sera approximée avec une autre qui ne sera pas forcément très proche à celle choisie avec Cross-Lib surtout si la machine a un nombre très limité de couleurs possibles. Par exemple la couleur bleue pourrait être rendue avec du vert sur une machine qui n'a pas cette couleur ou qui n'a pas assez de couleurs affichables en même temps.

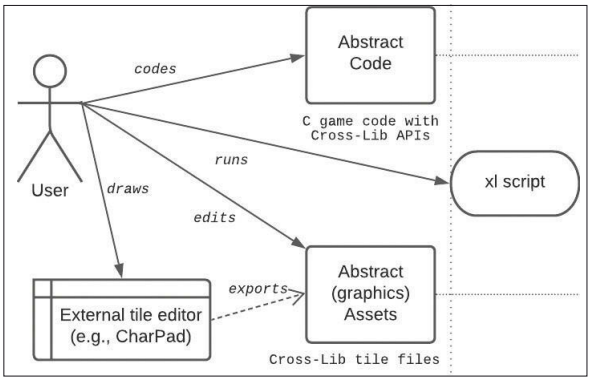


Figure 1

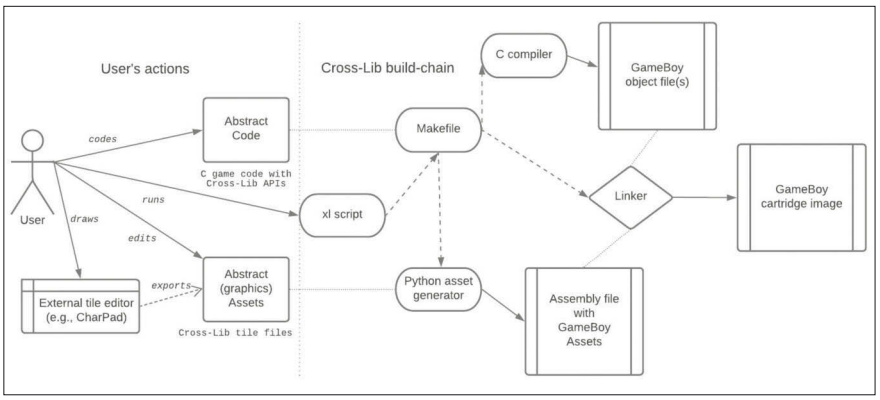


Figure 2

Il faut aussi remarquer que l'écran a une couleur d'arrière-plan qui sera fixée par le paramètre BACKGROUND_COLOR spécifié dans le fichier Makefile.<project> du projet <project> avec la couleur noire par défaut. Actuellement, les seules couleurs supportées pour l'arrière-plan ne sont que le noir (_XL_BLACK) et le blanc (_XL_WHITE).
Les couleurs du premier plan avec un arrière-plan noir (couleur par défaut) actuellement supportées sont :

```
_XL_WHITE, _XL_RED, _XL_GREEN, _XL_CYAN, _XL_YELLOW, _XL_BLUE ;
```

et avec un arrière-plan blanc :

```
_XL_BLACK, _XL_RED, _XL_GREEN, _XL_CYAN, _XL_YELLOW, _XL_BLUE.
```

Donc on ne supporte pas une couleur de premier plan qui coïncide avec celle de l'arrière-plan (qui n'a pas beaucoup d'intérêt).
Dans le futur, on prévoit davantage de couleurs, mais il ne faudra jamais oublier que les couleurs utilisées avec Cross-Lib seront toujours approximatives sur chaque machine différente, avec la couleur possible la plus proche sur la machine spécifique. Pour cette raison il faudra toujours choisir des couleurs très différentes si l'on veut être sûr qu'elles soient affichées avec une couleur réelle différente sur la plupart des machines.

10.3 TILES ET CARACTÈRES

Cross-Lib permet d'afficher deux types d'objets différents :

- *caractères* : caractères alphanumériques (dont on ne peut pas définir la forme, qui restera la même pour chaque machine sur tous les projets, mais pas forcément exactement la même sur toutes les machines) ;
- *tiles* : formes qu'on peut définir avec une liberté plus ou moins étendue selon la machine.

L'écran est vu comme une sorte de grille de taille donnée par

les *macros* XSize par YSize dans laquelle chaque « cellule » contient une des *tiles* ou caractères possibles.
Actuellement chaque tile et caractère ne peut avoir qu'une seule couleur de premier plan et la couleur du fond est partagée sur tout l'écran.

10.4 TILES

Les graphismes sur Cross-Lib sont exclusivement basés sur les tiles. Il existe principalement deux types de machines sur Cross-Lib : celles sans graphismes et celles avec graphismes. La différence principale entre les deux types de machines sera donnée par les « formes » possibles qu'on pourra donner aux tiles.
Dans le code, les tiles sont représentées par les entiers non-négatifs à 8-bit (uint8_t) donnés par les macros :

```
_TILE_0, _TILE_1, ...,_TILE_{N-1}
```

Actuellement N est égal à 26. Donc on a un nombre très limité de tiles.
Le code « ne connaît pas la forme » des tiles et celle-ci n'est même pas définie dans le code. Toutes les formes des tiles sont définies dans les répertoires dans src/<project>/tiles/.
Pour afficher la tile <tile> sur la « cellule » (x,y) avec la couleur de premier plan <color> on utilise

```
void _XL_DRAW(uint8_t x, uint8_t y, uint8_t tile, uint8_t color)
```

et pour effacer la « cellule » (x,y) :

```
void _XL_DELETE(uint8_t x, uint8_t y)
```

Par exemple, la ligne de code suivante

```
_XL_DRAW(12,5,_TILE_7,_XL_WHITE);
```

affichera la tile numéro 7, avec la couleur blanche de premier plan ayant pour coordonnées x=12, y=5 où la position initiale (0,0) correspond à l'angle supérieur gauche.

L'exemple tiles qui se trouve dans le répertoire src/examples/tiles montre comment utiliser _XL_DRAW et affiche les mêmes tiles sur tous les machines. Par exemple on peut le compiler pour le Commodore VIC 20 avec xl tiles vic20. Lorsqu'on lance cet exemple sur le Commodore VIC 20 on verra que certains tiles sont des *pre-shifted tiles*, c'est-à-dire qu'elles décrivent l'animation et le mouvement d'un personnage :

MACHINES SANS GRAPHISMES : pour les machines qui n'ont pas de mode graphique, les seules formes possibles pour les tiles sont celles des codes ASCII (ou de l'extension ASCII spécifique à chaque machine).
Dans ce cas-là, le code ASCII de chaque tile est défini dans le fichier src/<project>/tiles/ASCII /char_tiles.h. Par exemple src/snake/tiles/ASCII/char_tiles.h contient les définitions des « formes » des tiles pour le jeu *Cross Snake* pour toutes les machines qui n'ont pas de graphismes.

MACHINES AVEC GRAPHISMES : pour les machines avec de vrais graphismes possibles, on pourra définir la forme dans les moindres détails, pixel par pixel. La taille des tiles (nombre de pixels horizontaux et verticaux) dépend de la machine. Pour chaque taille de tiles possible, on définira une seule fois la forme qui sera partagée sur toutes les machines avec la même taille de tiles. Par exemple toutes les machines avec des tiles de 8 pixels par 8 pixels utilisent les mêmes définitions.
Le dossier src/<project>/tiles/ contient des dossiers pour chaque taille supportée. Actuellement on a :

- src/<project>/tiles/8x8/ : pour la plupart des machines ;
- src/<project>/tiles/6x8/ : pour la série Oric et des ordinateurs NTSC basés sur le RCA 1802 ;
- src/<project>/tiles/6x9/ : pour la plupart des ordinateurs PAL basés sur la puce RCA 1802 ;
- src/<project>/tiles/7x8/ : pour les Apple][et Apple//e en modalité HGR.

Par exemple src/bomber/tiles/8x8/ contient les définitions de tiles pour le jeu *Cross Bomber* pour toutes les machines qui ont de vrais graphismes avec tiles de taille 8x8.
Dans chacun de ces dossiers on trouve les fichiers TILE_0.txt, TILE_1.txt,… qui définissent la forme de chaque tile. Le format de chacun de ces fichiers est un CSV avec des chiffres en décimal ou hexadécimal (notation $XX) où chaque élément du CSV représente une ligne de la tile.
Pour créer de nouvelles formes, on peut :

1. dessiner les tiles avec des outils externes (comme *CharPad*) qui puissent exporter dans un format Assembleur 8-bit quelconque et les importer sur un projet ;
2. créer des fichiers de texte avec une « image » dont les pixels sont représentés par `#` pour le 1 (couleur du premier plan) et `.` pour le 0 (couleur de l'arrière-plan) et les importer sur un projet ;
3. éditer manuellement les fichiers TILE_0.txt, TILE_1.txt,...

OUTILS EXTERNES : pour produire des fichiers Assembleur qu'on pourra convertir dans le format Cross-Lib il faudra des outils qui exportent en format Assembleur sous forme de directives « .byte » ou « .decb » ou « fcb » ou « db ».

On peut utiliser :
VChar64 https://retro.moe/2015/02/10/vchar64-character-editor-for-the-commodore-64/
CharPad https://subchristsoftware.itch.io/charpad-free-edition
C64 Graphics Maker https://agpx.itch.io/c64-graphics-maker
Magellan https://magellan.js99er.net/
Pour importer sur un projet il suffira d'utiliser xl import <file> <project>.
Par exemple on peut créer un nouveau project foo et importer les tiles décrites par le fichier ./assets/examples/tile_sets/tiles/tiles_all.asm avec :

```
xl create foo
xl import ./assets/examples/tile_sets/tiles/tiles_all.asm foo
```

Il faut remarquer que cette méthode ne fera l'importation que pour les tiles en format 8x8 (qui couvre la plupart des cas). Pour les autres cas, il faudra adapter les tiles 8x8 ou éditer d'autres tiles.
Si l'on veut juste regarder les formes des tiles contenues dans

le fichier Assembleur on peut utiliser xl import <asm file>. Par exemple :

```
xl import ./assets/examples/tile_sets/tiles/tiles_all.asm
```

nous montrera les formes de tiles décrites avec des caractères `#` et `.`.
Il faut remarquer que xl import est capable de faire beaucoup plus : il peut faire du *ripping* et importer les tiles depuis du code source écrit en plusieurs dialectes Assembleurs et BASIC 8-bit en cherchant des patterns dans le texte du code. On trouve des exemples en plusieurs dialectes BASIC (y compris des exemples de BASIC 10-liner qui ont participé à la compétition BASIC 10-liner [bas10]) et Assembleur dans src/assets/examples/ .

DESSINER LES TILES COMME DU TEXTE : une autre solution est d'éditer les tiles dans des fichiers de texte avec `#` pour le 1 (couleur du premier plan) et `.` pour le 0 (couleur d'arrière-plan) et avec un maximum de 8 colonnes et un nombre libre de lignes.
Pour importer chaque tile pour chaque format possible on pourra simplement utiliser :

```
xl tile <tile text file> <project> <index>
```

qui importera une tile sur un projet pour le format du fichier de texte.
Par exemple on pourra importer une tile décrite par un fichier de texte comme celui-ci :

dans un projet foo qu'on a créé (par exemple avec xl create foo) comme tile numéro 7 (format 8x8 détecté automatiquement) avec :

```
xl tile ./assets/examples/single_tiles/tile_shape0.txt foo 7
```

Cross-Lib n'expose pas d'API pour les *sprites* (des objets qu'on peut placer sur l'écran avec une précision supérieure à celle des tiles et même au pixel près si on le désire), mais on peut les implémenter à travers des *pre-shifted tiles*, c'est-à-dire des tiles qui décrivent le mouvement et/ou l'animation d'un *sprite*. Les jeux *Cross Bomber*, *Cross Snake*, et *Cross Horde* utilisent des *pre-shifted tiles* que l'on peut voir avec xl show <project> par exemple :

```
xl show bomber
```

qui montrera les tiles utilisées dans le jeu *Cross Bomber*. Plus d'information sur xl show peut être affichée avec xl help show.

10.5 CARACTÈRES

Cross-Lib permet d'afficher sur toutes les machines que les caractères suivants :
- lettres majuscules de l'alphabet (`A`-`Z`),
- les chiffres (`0`-`9`) et
- le caractère espace.

Sur certaines machines on peut afficher les lettres minuscules. Les formes des caractères sont fixées pour chaque machine sur tous les projets.
Pour définir la couleur du texte que l'on veut afficher, on utilise :

```
void _XL_SET_TEXT_COLOR(uint8_t color)
```

La couleur est maintenue pour toutes les commandes qui affichent des caractères, mais pas forcement après des commandes pour afficher des tiles.
Pour afficher les lettres majuscules, les chiffres et l'espace on peut utiliser :

```
void _XL_CHAR(uint8_t x, uint8_t y, char ch) pour un seul caractère
```

et

```
void _XL_PRINT(uint8_t x, uint8_t y, char * str) pour une chaîne de caractères.
```

Pour afficher des nombres entiers non-négatifs à 16 bit (uint16_t) avec un nombre exact de chiffres donnés, on peut utiliser :

```
void _XL_PRINTD(uint8_t x, uint8_t y, uint8_t digits, uint16_t number)
```

Par exemple:

```
_XL_PRINTD(0,0,5,42U)
```

affichera 00042 dans la cellule (0,0), en haut à gauche.

11. INPUT, MANETTE OU CLAVIER

Selon la machine, l'input se fera soit à travers la manette (*joystick*) soit à travers le clavier (par défaut avec les touches `I`,`J`,`K`,`L` pour les directions et ESPACE pour *fire*).
Avant de commencer, il faudra toujours initialiser l'input avec

```
void _XL_INIT(void)
```

Les routines les plus importantes pour l'input sont :

```
uint8_t _XL_INPUT(void)
uint8_t _XL_LEFT(uint8_t input)
uint8_t _XL_RIGHT(uint8_t input)
uint8_t _XL_UP(uint8_t input)
uint8_t _XL_DOWN(uint8_t input)
uint8_t _XL_FIRE(uint8_t input)
uint8_t _XL_WAIT_FOR_INPUT(void)
```

Avec _XL_INPUT on peut lire l'input, et avec _XL_LEFT, _XL_RIGHT, _XL_UP, _XL_DOWN on peut l'interpréter.
Par exemple pour détecter le mouvement à gauche :

```
...
uint8_t input;
...
input = _XL_INPUT();
..
if(_XL_LEFT(input))
{
  // Handle left direction
  ...
}
```

Avec _XL_WAIT_FOR_INPUT on bloque l'exécution jusqu'à la pression d'une touche (une touche du clavier ou le bouton fire pour les machines avec joystick).

12. EFFETS SONORES

La version actuelle de Cross-Lib ne prévoit qu'un nombre très limité d'effets sonores prédéfinis.
Trois effets sonores très courts sont produits par :

```
void _XL_PING_SOUND(void)
void _XL_TICK_SOUND(void)
void _XL_TOCK_SOUND(void)
```

Un effet plus long est produit par :

```
void _XL_ZAP_SOUND(void)
```

On peut produire deux bruits différents avec :

```
void _XL_SHOOT_SOUND(void)
void _XL_EXPLOSION_SOUND(void)
```

13. AUTRES FONCTIONS UTILES

Pour pouvoir coder un jeu avec le même code il faut aussi d'autres fonctions comme celle qui permet de mettre en pause le jeu et une fonction qui puisse générer des nombres aléatoires avec exactement le même range à 15 bits (0-32676) sur toutes le machines.

13.1 PAUSES

PAUSE LONGUE : pour mettre en pause un jeu pour un nombre de secondes, on peut utiliser :

```
_XL_SLEEP(uint8_t sec)
```

À noter que sur certaines machines la durée d'une seconde peut être une approximation.
MICRO-PAUSE : on peut mettre en pause un jeu pour un « nombre de boucles » avec :

```
_XL_SLOWDOWN(uint16_t loops)
```

où chaque boucle (qui correspond à une boucle vide de code C) aura une durée différente sur chaque machine et qu'il faudra donc multiplier par un facteur de proportionnalité qui est donné par la macro_XL_SLOWDOWN_FACTOR, qui fera en sorte d'avoir soit la même pause sur toutes les machines ou une pause qui dépendra par exemple de la taille de l'écran si l'on veut qu'un jeu tourne moins rapidement sur certaines machines. La valeur de _XL_SLOWDOWN_FACTOR est définie dans le fichier Makefile.<project> pour chaque projet.

13.2 NOMBRES ALÉATOIRES

En ANSI C la fonction rand() qui est incluse dans stdlib.h n'a pas un range exactement défini dans le standard et elle n'est pas présente sur tous les dev-kits 8-bit.
Pour cette raison Cross-Lib dispose d'une fonction qui produit des nombres pseudo-aléatoires non-négatifs à 15-bit (0-32767) sur toutes les machines :

```
uint16_t _XL_RAND(void)
```

14. UNE PETITE ANIMATION

Avec les commandes vues jusqu'ici, on peut déjà créer un jeu embryonnaire avec un petit bonhomme contrôlé par la manette ou le clavier. Un tel exemple est déjà présent dans les exemples qu'on trouve dans le répertoire src/examples/animate/ :

```
#include "cross_lib.h"

#define MIN_X 1
#define MAX_X (XSize-2)
#define MIN_Y 1
#define MAX_Y (YSize-2)
#define DOWN_TILE  _TILE_0
#define UP_TILE    _TILE_1
#define RIGHT_TILE _TILE_2
#define LEFT_TILE  _TILE_3
#define FIRE_TILE  _TILE_4

int main(void)
{
 uint8_t x;
 uint8_t y;
 uint8_t input;
 uint8_t tile;

 _XL_INIT_GRAPHICS();
 _XL_INIT_SOUND();
 _XL_INIT_INPUT();

 _XL_CLEAR_SCREEN();

 x = XSize/2;
 y = YSize/2;
 tile = DOWN_TILE;
 hile(1)
 {
     _XL_SET_TEXT_COLOR(_XL_WHITE);
     _XL_PRINTD(0,0,3,x);
     _XL_PRINTD(5,0,3,y);

     input = _XL_INPUT();
     if(_XL_UP(input))
     {
       _XL_DELETE(x,y);
       tile = UP_TILE;
       —y;
     }
     else if (_XL_DOWN(input))
     {
       _XL_DELETE(x,y);
       tile = DOWN_TILE;
       ++y;
     }
     else if (_XL_LEFT(input))
     {
       _XL_DELETE(x,y);
       tile = LEFT_TILE;
       —x;
     }
     else if(_XL_RIGHT(input))
     {
       _XL_DELETE(x,y);
       tile = RIGHT_TILE;
       ++x;
     }
```

```
    else if(_XL_FIRE(input))
    {
      _XL_DRAW(x,y,FIRE_TILE,_XL_WHITE);
      _XL_EXPLOSION_SOUND();
      _XL_SLOW_DOWN(16*_XL_SLOW_DOWN_FACTOR);
    }

   if((y>=MIN_Y)&&(y<=MAX_Y)&&(x>=MIN_X)&&(x<=MAX_X))
    {
      _XL_DRAW(x,y,tile,_XL_WHITE);
    }
    else
    {
      _XL_DELETE(x,y);
      _XL_ZAP_SOUND();
      _XL_SLOW_DOWN(16*_XL_SLOW_DOWN_FACTOR);
      x = XSize/2;
      y = YSize/2;
    }
    _XL_SLOW_DOWN(4*_XL_SLOW_DOWN_FACTOR);
  }
  return EXIT_SUCCESS;
}
```

15. COMMENT CODER POUR TOUTES LES MACHINES ?

Cross-Lib supporte des machines avec des ressources hardware très différentes : taille de l'écran différente, présence ou absence de graphismes, d'effets sonores possibles, couleurs, etc. Les API de Cross-Lib ne suffissent pas pour avoir du code qui s'adaptera à tous les cas possibles. Par contre Cross-Lin expose des macros qui permettent d'écrire du code capable de s'adapter à toutes les machines.

Par exemple pour avoir un jeu capable de s'adapter à toutes les tailles d'écran il faudra utiliser des fractions entières de macros XSize et YSize pour s'adresser aux « cellules » de l'écran. Dans certains cas il faudra utiliser des directives #if pour gérer des écrans avec des tailles minuscules (comme c'est le cas pour certaines consoles *hand-held* ou ordinateurs moins équipés ou calculatrices).

Pour avoir un jeu qui marche aussi bien sur une machine avec couleurs que sur une sans couleurs il faudra soit éviter que le jeu utilise les couleurs dans sa logique ou prévoir des directives comme #if defined(_XL_NO_COLOR) pour implémenter une partie de la logique pour les machines avec couleurs et une autre pour les machines sans couleurs. Idéalement on veut minimiser ou ne pas avoir de directives, mais parfois on n'a pas de choix si l'on veut un jeu qui tourne bien sur toutes les machines.

Cross-Lib expose les macros qui décrivent les détails de ce que Cross-Lib prévoit pour chaque machine pour permettre au développeur de gérer les différences. Si l'on veut connaître ce que Cross-Lib a implémenté pour une machine précise, il suffira de regarder et compiler l'exemple target (dont le code se trouve dans src/examples/target/) pour la machine qui nous intéresse. Le code de cet exemple utilise tous les macros disponibles pour gérer les différences entre les machines : XSize, YYize, _XL_TILE_X_SIZE, _XL_TILE_Y_SIZE, _XL_NUMBER_OF_TILES, _XL_NO_COLOR, _XL_NO_TEXT_COLOR, _XL_NO_UDG, _XL_NO_JOYSTICK, _XL_NO_COLOR, _XL_NO_SOUND, _XL_NO_CAPITAL_LETTERS.

Par exemple :

```
xl target
```

produira un fichier exécutable pour le terminal natif que l'on pourra lancer avec

```
xl run target
```

qui montre que pour le terminal natif, Cross-Lib prévoit un écran 80x24 avec des tiles « ASCII » (sans graphismes) et avec tous les autres *flags* actifs à l'exception des effets sonores et de la manette (*joystick*) :

```
TARGET INFORMATION
XSIZE 32  YSIZE 24
TILES 26  8X8
GRAPHICS        ON
COLOR           ON
TEXT COLOR      ON
JOYSTICK        ON
SOUND           ON
SMALL CHARS     ON
```

Par contre avec

```
xl target msx
```

on produira l'image d'une cartouche (*rom*) pour les ordinateurs *MSX* qui nous montre que pour ces ordinateurs, Cross-Lib prévoit un écran 32x24 avec des tiles 8x8 (donc avec graphismes) et avec tous les *flags* actifs.

```
TARGET INFORMATION
XSIZE 80  YSIZE 24
TILES 26  ASCII
GRAPHICS        OFF
COLOR           ON
TEXT COLOR      ON
JOYSTICK        OFF
SOUND           OFF
SMALL CHARS     ON
```

CONCLUSION

Les avantages les plus importants de Cross-Lib sont : d'être WORA et de supporter 200 machines. Tout cela a été possible parce que Cross-Lib n'expose qu'une partie des caractéristiques de chaque machine et cela a des limitations (nombre des tiles, effets sonores, etc.). Ces contraintes pourraient être partiellement levées dans les versions futures. Un autre aspect qu'on prévoit d'améliorer est l'installation des dépendances (compilateurs, outils), qui pourrait ne plus être nécessaire avec une image Docker.
A vous de coder !

Bibliographie

[ack] **ACK**, Tanenbaum A., Jacobs C. *et al.*, pour l'Intel 8088, PDP 11, CP/M-80,etc., https://github.com/davidgiven/ack

[app] **Cross Chase: A Massively 8-bi Multi-System Game**, Caruso F., *Call-A.P.P.L.E.*, vol. 28, No. 1, Feb 2018, pages 31-33, http://www.callapple.org

[bas10] **BASIC 10-liner competition**, Kanold G. : https://gkanold.wixsite.com/homeputerium/

[east] **Discovering Eastern Europe PCs by Hacking Them… Today**, Bodrato S., Caruso F., Cignoni G., *IFIP International Conference on the History of Computing (HC)*, Sep 2018, Poznan, Poland. pp.279-294

[c8b] **8-bit C**, Caruso F., https://github.com/Fabrizio-Caruso/8bitC

[cc6303] **CC6303**, Cox A., Compilateur C pour le Motorola 6800 et 6803, https://github.com/EtchedPixels/CC6303

[cc65] **CC65**, Compilateur C pour l'architecture MOS 6502, https://sourceforge.net/projects/cc65/

[ceo] **Cross Chase**, Caruso F., *CEO-MAG*, No. 327-328, Juillet-Août 2017, https://ceo.oric.org/

[cmoc] **CMOC**, Sarrazin P., Compilateur C pour le Motorola 6809, https://perso.b2b2c.ca/~sarrazip/dev/cmoc.html

[effc] **Efficient C Code for 8-bit Microcontrollers**, Jones N., https://barrgroup.com/embedded-systems/how-to/efficient-c-code

[m20] **Z8KGCC**, Groessler C., GCC pour l'Olivetti M20, http://www.z80ne.com/m20/sections/download/z8kgcc/z8kgcc.html

[lcc1802] **LCC1802**, Rowe B., Compilateur C pour le RCA 1802, https://github.com/bill2009/lcc1802

[st] **GCC**, Rivière V., version pour l'Atari ST, http://vincent.riviere.free.fr/soft/m68k-atari-mint/

[ti] **GCC**, (*Insomnia*), version pour le TI99/4A, http://atariage.com/forums/topic/164295-gcc-for-the-ti

[trse] **TRSE**, Groeneboom N. E. *et al.*, dev-kit basé sur un dialecte Pascal pour plusieurs machines 8-bit et 16-bit, https://lemonspawn.com/turbo-rascal-syntax-error-expected-but-begin/

[un8] **8-Bit Unity**, Beaucamp A. (*8-bit Dude*), http://8bit-unity.com/

[vbcc] **VBCC**, Barthelmann V., Compilateur C pour le BBC Micro, le BBC Master, l'Amiga, etc., http://www.compilers.de/vbcc.html

[xlib] **Cross-Lib**, Caruso F., https://github.com/Fabrizio-Caruso/CROSS-LIB/

[xtc68] **XTC68**, Hudson J., Compilateur C pour le Sinclair QL, https://github.com/stronnag/xtc68

[z88dk] **Z88DK**, Compilateur C pour les architectures Zilog 80 et Intel 8080, https://github.com/z88dk/z88dk

INSTALLATION SOUS UBUNTU 20.04

Sous Ubuntu 20.04 (aussi utilisable sous Windows 10 via Windows Subsystem for Linux), on pourrait installer tout ce qu'il est nécessaire pour compiler pour la majorité des machines basées sur le MOS 6502 avec les commandes suivantes :

```
sudo apt-get update -y
sudo apt install make
sudo apt-install python
```

Pour installer une version récente de CC65 (en 2021) :

```
sudo apt-install cc65
```

Pour Cross-Lib, il faut tout simplement le télécharger avec

```
git clone https://github.com/Fabrizio-Caruso/CROSS-LIB.git
```

Pour émuler les Commodore 8-bit on peut installer Vice avec

```
sudo apt-install vice
```

et copier les roms (que l'on trouve « ailleurs ») dans les répertoires sous /usr/lib/vice/
On pourra compiler les jeux et les exemples distribués avec Cross-Lib avec

```
xl <projet> <target>
```

Donc pour compiler *Cross Snake* pour le VIC 20 il suffira :

```
xl snake vic20
```


PROGRAMMEZ!

Programmez! hors-série n°6 - HIVER 2021-2022
Directeur de la publication & rédacteur en chef
François Tonic
ftonic@programmez.com

Tech lead du numéro : **Franck Dubois**

Contacter la rédaction
redaction@programmez.com

Expert en brèves : **Louis Adam (ZDnet.fr)**

Les contributeurs techniques
J-C Ize — **M. Hage Chahine**
F. Larivé — **G. Bersegeay**
F. Rival — **J. Lizard**
F. Dubois — **M. Kruzik**
J-M Papillon — **F. Caruso**

Couverture : © **Matrix, D.R.**

Maquette : **Pierre Sandré**

Marketing – promotion des ventes
Agence BOCONSEIL - Analyse Media Etude
Directeur : Otto BORSCHA
oborscha@boconseilame.fr
Responsable titre : Terry MATTARD
Téléphone : 09 67 32 09 34

Publicité
Nefer-IT - Tél. : 09 86 73 61 08
ftonic@programmez.com

Impression : **SIB Imprimerie, France**

Dépôt légal : **A parution**

Commission paritaire
1225K78366

ISSN : **2279-5001**

Abonnement
Abonnement (tarifs France) : 49 € pour 1 an, 79 € pour 2 ans. Etudiants : 39 €. Europe et Suisse : 55,82 € - Algérie, Maroc, Tunisie : 59,89 € - Canada : 68,36 € - Tom : 83,65 € - Dom : 66,82 €.
Autres pays : consultez les tarifs sur www.programmez.com.
Pour toute question sur l'abonnement :
abonnements@programmez.com
Abonnement PDF monde entier : 39 € pour 1 an.
Accès aux archives : 19 €.

Nefer-IT
57 rue de Gisors, 95300 Pontoise France
redaction@programmez.com
Tél. : 09 86 73 61 08

5e édition

Suivez le Guide

• Sécuriser le télétravail – Gestion de crise – Ransomware
Maitriser le vocabulaire de la cybersécurité : zero-trust, Sase, Bug Bounty etc

Professionnels : trouvez les informations de référence pour la Cybersécurité
de votre entreprise, ainsi que la présentation des acteurs du marché et des organismes.

Consultez le Guide en ligne :
www.solutions-numeriques.com/securite/

034SLG

softfluent

Rejoignez **l'ESN** créée par des **développeurs** pour les Développeurs

Vous possédez une ou plusieurs de ces compétences ?

- Azure DevOps
- Cloud
- SQL Server
- Angular
- React
- C#
- CI/CD
- .NET Core
- ASP.NET MVC
- Microservices

Contactez nous !

Retrouvez-nous sur notre page carrière : **softfluent.fr/nous-rejoindre**

Ou contactez Marine, en charge du recrutement chez SoftFluent :

06 69 26 27 87

marine.genetay@softfluent.com

- Conseil
- Expertise
- Partage

softfluent.fr