
LE JEU
DU CODE
RONAN LEBAS

Le jeu du code :
En quoi le *modding* est-il une forme d'expression ?

Par Ronan LEBAS

Écrit réalisé dans le cadre du
PACE 2022-2023 de Télécom Paris

Table des matières

Introduction

Le jeu vidéo a su au cours des années s'imposer comme un loisir de référence, notamment parmi les jeunes populations. La popularité grandissante de cette pratique a donc permis de créer une véritable industrie : pour satisfaire le public, il faut développer des jeux convaincants, ce qui n'est pas chose facile. Intéressé depuis longtemps par cet univers, je me propose d'étudier ce sujet, en analysant un cas concret : le mien. Mon but est de mettre en évidence le fait que la création de jeu vidéo est une forme d'expression à part entière - et requiert donc une réflexion au-delà de celle demandée par l'aspect pratique de la programmation. Je souhaite également expliquer ce qu'est le *modding* et ce qu'il permet. Nous mènerons notre étude en trois parties : tout d'abord un bref historique du développement de jeu et une introduction au concept de *modding*, puis une mise en contexte retraçant ma découverte des jeux vidéo et de la programmation, et enfin une analyse précise de mon processus de création par le biais du *modding*. Par ailleurs, je tiens à préciser que je ne me considère pas comme un véritable expert du domaine, et que ce qui suit n'est qu'un point de vue personnel.

Historique du développement de jeux et du *modding*

Histoire et progrès technologique

Le concept de jeu vidéo a eu bien des définitions, reposant pour la plupart sur les moyens techniques employés, et de ce fait, excluant beaucoup de jeux. Aujourd'hui, on appelle jeu vidéo tout dispositif électronique visant à une interaction homme-machine ludique et affichant un retour sur un écran. Le jeu vidéo a connu des débuts modestes : alors qu'il n'était pas encore formalisé, les chercheurs des années 1950 travaillant sur le développement de l'ordinateur réalisaient des prototypes de jeux extrêmement simplistes pour démontrer la faisabilité de certaines choses et la polyvalence des ordinateurs. Le premier jeu réalisé "en tant que jeu" est *Tennis for Two*, un jeu

qui opposait deux personnes devant se renvoyer un point de lumière sur l'écran d'un oscilloscope.

Photographie de 1959 du matériel de Tennis For Two.
https://fr.wikipedia.org/wiki/Tennis_for_Two

Au début des années 1970, Atari, une des premières entreprises spécialisées dans le jeu vidéo, développe *Pong*, le premier jeu à connaître un véritable succès, d'abord grâce aux bornes d'arcades, puis avec des consoles de salon. Les bases du jeu vidéo sont posées pour les années qui suivent : manettes, consoles de jeu destinées à être reliées à un écran de télévision, utilisation de l'ordinateur comme adversaire et système de score - ceci permettra d'assurer la survie des jeux vidéo. La décennie suivante, le marché s'emballe, voyant de nombreux modèles apparaître, et en 1983 survient le krach de l'industrie. Cet effondrement freine le développement des jeux vidéo aux États-Unis, permettant ainsi au Japon de devenir le leader du marché, et grâce au progrès technologique toujours plus important, les jeux spécifiquement conçus pour console (et non pour borne d'arcade), tels que *Super Mario Bros.* pour NES, naissent et la majorité des genres connus aujourd'hui apparaissent. Le milieu des années 1995 marque une révolution car c'est le début des jeux en trois dimensions, accompagnés par une complexification des scénarios et des mécaniques de jeu. L'avènement d'Internet et des appareils mobiles achève de transformer le jeu vidéo pour en faire la pratique que nous connaissons

aujourd'hui, en amenant le jeu en ligne et la possibilité de jouer partout et tout le temps sur son smartphone. On peut catégoriser les créateurs de jeux en trois générations suivant ces évolutions technologiques[1] : les développeurs de la "première génération" ont codé des jeux essentiellement seuls, car ils étaient les pionniers. La "deuxième génération", autodidacte comme la première et née dans les années 1990, est à l'origine de la plupart des studios reconnus actuellement. La "dernière", enfin, sort d'écoles spécialisées dans le domaine et possède donc une formation axée sur le jeu vidéo, contrairement aux deux générations précédentes, et la motivation des développeurs de la "troisième génération" provient des jeux auxquels ils ont joué dans leur enfance.

Définition de *modding*

Ainsi, alors qu'elle était un artisanat à ses débuts, la création de jeux vidéo a connu une expansion très importante, tant au niveau du public que des personnes y travaillant pendant son industrialisation. Mais cet artisanat originel a continué d'exister, sous la forme de ce que l'on appelle le *modding*, c'est-à-dire l'action de modifier un jeu en ajoutant ou modifiant des fonctionnalités. Pour *modder* un jeu, il faut bien sûr ne pas appartenir au studio de développement du jeu, car les mises à jour apportées à un jeu ne sont pas considérées comme des *mods*, qui doivent donc ne pas être officiels. Souvent, le *modding* est vu comme une activité de loisir, car non rémunéré - les *mods* sont généralement mis à disposition sur Internet gratuitement - et les *moddeurs* sont souvent seuls pour programmer leurs modifications, bien que, dans le cadre de projets importants, des équipes peuvent se former. Enfin, les *mods* sont le meilleur moyen d'exprimer la créativité d'un joueur dans un jeu, bien que programmer des modifications ne soit pas à la portée de tout le monde au premier abord. Afin de pousser cet aspect de joueur créateur, certains jeux prévoient même des supports pour aider à la réalisation de *mods*, car cela permet de rallonger la durée de vie du jeu en ajoutant du contenu fait par la communauté.

[1] François Rouet, "La création dans l'industrie du jeu vidéo", *Culture Études*, N°1, 2009, pp 1-16

Vers une création de la part du joueur ?

Cette pratique s'inscrit dans une dynamique plus large : le joueur est également créateur du jeu. Cela passe tout d'abord par les jeux qui laissent des choix aux joueurs, n'imposant alors pas un seul scénario. Mais il existe aussi des jeux - de construction par exemple - qui donnent des outils au joueur pour exprimer sa créativité, même si, dans ces cas, tout ce que pourra imaginer le joueur a déjà été "prévu" par les développeurs. De manière plus surprenante, de plus en plus de studios ont recours à l'accès anticipé, *Early access* en anglais. Cela consiste à vendre le jeu alors qu'il n'est pas fini, à un prix bien inférieur au prix final, en clamant l'accès anticipé.

Titre de la catégorie "Accès Anticipé" sur Steam, la principale plateforme d'achat en ligne de jeux vidéo.

Ainsi, les joueurs achètent un jeu souvent rempli de bugs mais peu cher, et reçoivent des mises à jour gratuites régulièrement durant le développement du jeu. Pour les développeurs, cela permet d'obtenir des revenus plus tôt, et cela offre une quantité de testeurs énorme, largement plus importante que si les développeurs devaient tester eux-mêmes leur jeu ou embaucher des testeurs. Par ailleurs, les tests sont réalisés en "condition réelle" puisque ce sont des joueurs "normaux" qui jouent - les tests ne sont pas biaisés : un développeur peut oublier de tester une fonctionnalité qu'il considère valide, orientera ses tests selon son expérience de programmeur, et se rendra moins bien compte de l'équilibrage du jeu. De ce fait, les bugs sont détectés plus vite et en plus grand nombre, l'équilibrage est mieux réussi, et des fonctionnalités désirées par les joueurs mais non prévues à la base peuvent être ajoutées. Il en résulte donc un jeu mieux réussi que s'il avait été développé sans l'accès anticipé. Cet accès anticipé est essentiellement utilisé pour ce qu'on appelle les "jeux indés", abréviation de "jeux indépendants", c'est-à-dire qu'ils sont développés par des petits studios souvent peu connus, contrairement aux jeux triple-A, produits par des grandes entreprises qui éditent des séries de jeux à succès. Les développeurs des jeux triple - A, quand ils sortent un nouveau jeu, savent qu'il sera beaucoup vendu, car il reprend les codes d'une lignée qui a

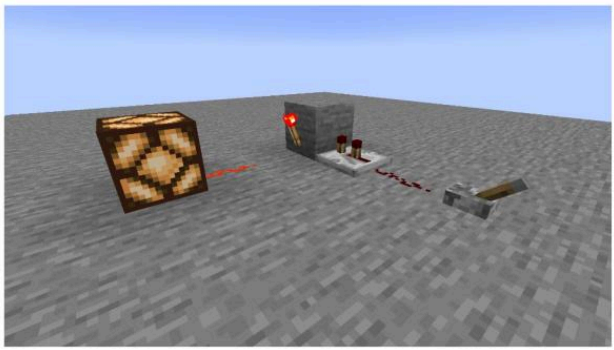
bien fonctionnée, et donc le recours à l'accès anticipé est inutile et ferait probablement perdre de l'argent.
Discutons maintenant d'un exemple de jeu où l'intégration des joueurs a causé des problèmes. Aux alentours de 2015, une équipe lance le développement d'un nouveau jeu[2] de la licence *Sword* (le nom est modifié dans l'article étudié pour clause de non-divulgation). Du fait d'un piratage important sur les précédents jeux, l'équipe de développement a dû trouver de nouvelles méthodes de travail, car simplement vendre un jeu n'était, pour eux, plus rentable. Le studio a donc décidé d'adopter une forme commerciale de plus en plus utilisée aujourd'hui - et qui s'est montrée au grand jour notamment avec le jeu *Fortnite*, résumée par un directeur de département de la société ainsi : "C'est pour ça que nous on est passés de deux à trois métiers ; ceux qui font le jeu, ceux qui le vendent, et ceux qui l'opèrent". Ce modèle part de ce principe : pour gagner de l'argent plus longtemps et pour que le jeu soit durable, il faut miser sur la vie du jeu après le lancement, c'est-à-dire en publiant des mises à jour ou en vendant du contenu optionnel, par exemple des objets cosmétiques. C'est la transformation du jeu vidéo, "d'industrie de produit [...] en industrie de service". Pour se renouveler et changer de philosophie, le studio a décidé d'utiliser l'*open-dev*, donc intégrer les joueurs dans le processus de développement. Il a été décidé de collaborer avec des *VIPs*, des joueurs reconnus dans la communauté des jeux *Sword*, disposant d'une expertise et d'une communauté. Les *VIPs* avaient accès à pratiquement toutes les informations, et étaient chargés d'apporter des retours pour veiller à l'équilibrage par exemple, mais surtout, ils étaient un moyen privilégié de promouvoir le jeu avant sa sortie, car ils avaient une audience importante et une influence directe sur leur communauté. Malgré la construction d'une relation entre l'équipe de développement et les joueurs *VIP*s, certains problèmes ont émergé du fait de la différence de statut entre les deux groupes, les premiers étant payés contrairement aux autres (et donc de l'absence d'autorité sur les *VIP*s) et de sujets de désaccord. Principalement, les *VIP*s souhaitaient des outils facilitant le *modding* et donc une grande liberté pour *modder*, car c'est selon eux ce qui assure la longévité du jeu, et lui permet d'augmenter sa qualité. En revanche, le studio n'avait pas prévu à la base de laisser autant de liberté aux joueurs, et ce sujet - très important à mon avis - a été le point de départ des conflits, menant à une mise à l'écart des *VIP*s, et finalement un arrêt complet de la collaboration. Nous pouvons ainsi comprendre que, même si l'avis des joueurs

[2] Mathieu Cocq, "L'organisation et l'exploitation du travail des joueurs", *Réseaux*, N°213, 2019, pp 111-137

est essentiel pour faire un bon jeu, la manière d'écouter cet avis est fondamentale pour le bon déroulement du développement du jeu.

Un exemple : *Minecraft*

L'un de ces jeux connus pour ses *mods* est *Minecraft*. En effet, avec des dispositifs mis en place par la communauté pour faciliter la création de *mods*, et plus d'un millier de *mods* développés, ce n'est pas un aspect sous-exploité du jeu. Pour présenter un peu ce jeu, nous pouvons dire que *Minecraft* est une référence dans le monde des jeux de type *sandbox*, i.e. "bac à sable", c'est-à-dire des jeux qui ne comportent pas forcément d'objectif défini, et qui se basent sur la créativité du joueur qui modifie son environnement à l'aide d'outils. *Minecraft* est une référence, et ce pour plusieurs raisons :
Tout d'abord, sa très grande popularité : avec plus de 200 millions d'exemplaires vendus, c'est le jeu vidéo le plus vendu de tous les temps. Pendant sa seconde campagne pour l'élection présidentielle, Emmanuel Macron a même lancé un serveur Minecraft représentant un "QG de campagne virtuel" - c'est dire à quel point ce jeu semble représenter toute une génération. Selon moi, c'est aussi une référence car c'est un jeu "bac à sable" à plusieurs niveaux. Je vais tenter d'expliquer cet avis : le premier niveau est la possibilité d'exprimer sa créativité à travers les blocs, on peut construire des bâtiments, des paysages et représenter à peu près tout grâce aux blocs, c'est l'objectif premier du jeu, c'est la façon de jouer la plus répandue chez les plus jeunes, et c'est la première chose à laquelle on pense quand on évoque *Minecraft*. Le deuxième niveau est la possibilité de réaliser des circuits de "redstone", forme d'électronique qui permet de faire des systèmes automatisés pouvant accomplir différentes tâches normalement effectuées par le joueur.



Exemple de "circuit de redstone" :
La lampe est allumée si et seulement si le levier est désactivé, c'est une porte logique "non". Il est possible de réaliser des circuits très complexes, comme des calculatrices.

Les éléments sont simples et leur utilisation facile à comprendre, et pourtant la créativité des joueurs peut mener à des réalisations surprenantes, comme des calculatrices quatre opérations totalement fonctionnelles. Le troisième niveau, lui, ne se situe plus qu'à moitié dans le jeu, ce sont les commandes et les *data packs* que nous aborderons plus tard, et le dernier niveau, le plus éloigné du jeu "normal", est celui des *mods*. Ce sont ces quatre niveaux d'expression de créativité qui, selon moi, font de *Minecraft* la référence des jeux *sandbox*.

Comment suis-je tombé dans le *modding* ?

C'est donc pour cette polyvalence que j'apprécie *Minecraft*. En particulier, une des premières choses qui m'ont frappées en découvrant le jeu, c'est le nombre incroyable de *mods* existants, introduisant chacun un univers connu, par exemple *Star Wars* ou bien *Harry Potter*, ou quelque chose directement inventé par le *moddeur*. L'impression qui en découle est : "Tout est possible". Si un monde *Minecraft* est si grand (8 fois la surface de la Terre selon certains calculs) qu'il peut sembler infini, le réel "infini" de ce jeu réside bien dans le nombre de *mods* qui existent, et qui restent à faire. Je vous propose maintenant de revenir sur ma rencontre avec ce jeu, les jeux vidéo en général et la programmation, car ils sont, pour moi, intimement liés.

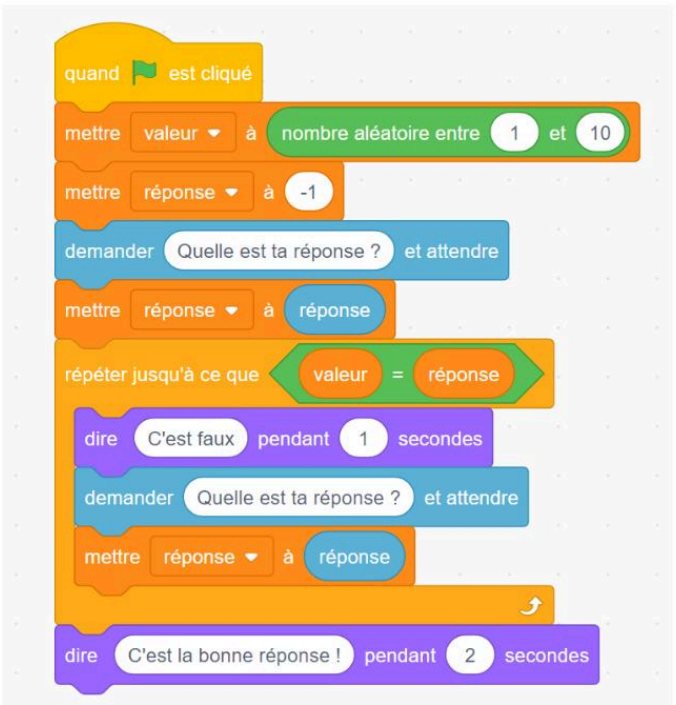
Du jeu sur console aux *mods* de jeu PC

Comme la plupart des enfants recevant leurs premiers jeux vidéo, à l'époque, je ne m'étais pas posé de questions sur la façon dont ils étaient faits. J'ai d'abord eu des jeux sur console de salon puis sur console portable. Comme la télévision et les autres objets modernes, ils étaient pour moi les fruits d'une science complexe et que je ne pouvais pas comprendre. Puis j'ai découvert *Minecraft*, par le biais de vidéos dans lesquelles l'auteur présentait des *mods*, ce qu'ils comportent et leur utilisation. C'est avec ces vidéos, présentant des *mods* sur des univers de fiction complètement indépendants de *Minecraft*, que j'ai compris l'immense diversité du contenu existant, et ces modifications si puissantes ont d'ailleurs été pour moi le seul intérêt de ce jeu pendant longtemps. Plus tard, j'ai tenté d'installer des *mods* sur l'ordinateur familial, sans pour autant y arriver, car la polyvalence des *mods* a une contrepartie : coder des *mods* nécessite bien sûr de l'expérience et du talent en programmation, mais les installer, même si c'est beaucoup plus simple, n'est pas non plus une opération triviale. En effet, comme on modifie le jeu en lui-même, il faut des connaissances en informatique et pas seulement dans le jeu. Malgré cela, j'ai tout de même réussi à installer des *mods* lors de ma seconde tentative quelques années plus tard, pour finalement être déçu par des bugs présents dans ces *mods* qui ont corrompu la sauvegarde. Cette expérience m'a quelque peu dégoûté des *mods*, bien que ma responsabilité quant au choix des *mods* et de leur compatibilité entre eux et avec la version du jeu utilisée n'est pas à nier. C'est notamment cet épisode qui m'a encouragé à me tourner, plus tard, vers les *data packs*, du fait de leur simplicité d'installation, et de leur incapacité à faire planter le jeu comme un *mod* pourrait le faire.

Mes débuts dans le monde de la programmation

En parallèle, et au fur et à mesure de ma scolarité, la programmation est venue à ma rencontre. En effet, c'est lors d'un cours de technologie au collège que l'on nous a montré *Scratch*, un langage de programmation graphique permettant de s'initier au concept de code. Pas besoin d'apprendre des mots-clés ou d'écrire des programmes, il suffit d'assembler des briques de couleur à l'aide de la souris dans une interface graphique. Chacune de ces briques correspond à une instruction qu'on trouverait habituellement dans un langage de programmation. Ainsi, on ne peut pas faire d'erreur de syntaxe car

les briques ne s'emboîtent que si ce que l'on tente de faire est juste, on comprend facilement la signification des instructions car elles sont traduites, et il y a même un système de "lutins" : des personnages qui apparaissent dans une fenêtre et que l'on peut contrôler à l'aide d'instructions du type "avance de 10 unités vers la gauche". La découverte en classe de ce langage suivait un tutoriel pour réaliser le jeu *Pong*, dont nous avons déjà parlé, dans une version pour un joueur seul et dans laquelle il suffisait de renvoyer la balle contre le mur. J'ai tout de suite été fasciné par les possibilités de cet outil, je l'ai vite assimilé, et je me rappelle avoir modifié le programme *Pong* demandé pour en faire un jeu plus amusant, avec un adversaire automatique, un système de meilleur score, et un mode multijoueur pour affronter d'autres élèves. Je crois pouvoir dire que c'est là que mon envie de réaliser des jeux vidéo est née. J'ai ensuite fait quelques autres jeux ou animations mineurs avec *Scratch*, mais rien de plus.



Exemple classique de programme réalisé avec Scratch *:*
L'ordinateur tire un nombre aléatoire, et l'utilisateur essaye de le deviner en proposant des nombres jusqu'à trouver la bonne réponse.

Programmer avec des mots

Au lycée, j'ai obtenu comme les autres élèves une calculatrice graphique, la *TI-82 Advanced*, servant à calculer bien sûr, mais aussi à réaliser quelques programmes dans le cadre du cours de mathématiques, par exemple une fonction prenant comme arguments les coordonnées de trois points et renvoyant la nature du triangle associé (rectangle, isocèle…). Ces programmes étaient donnés par le professeur et traduits dans les langages correspondant aux deux types de calculatrices répandus. Le but était de nous familiariser avec la logique de la programmation, tout comme, en terminale, les ateliers de programmation sur ordinateur en langage Python - bien plus utilisé que les langages propres aux calculatrices TI et Casio. Si ces ateliers sur ordinateur ne m'ont pas vraiment inspiré, ce ne fut pas le cas des programmes pour calculatrices. En classe de première, il y a dans le cours de mathématiques une technique de résolution pour certaines équations qui est très "algorithmique" : le nombre de solutions de l'équation et leurs valeurs sont donnés explicitement par une formule, on peut donc résoudre facilement ces équations de manière algorithmique. Je me suis dit que l'apprentissage de cette méthode était la bonne opportunité pour explorer la programmation par moi-même, la "vraie" programmation avec des mots que l'on écrit et qui me paraissait, à première vue, bien plus compliquée que coder en utilisant les blocs de *Scratch*. C'était le début de ma découverte de la programmation, constituée de beaucoup de temps passé à comprendre la syntaxe du langage, le principe des structures de contrôle de l'exécution bien connues en informatique, telles que *if-else* ou *for* par exemple. J'ai donc réalisé des petits programmes sur ma calculatrice mettant en application le cours de mathématiques, pour apprendre la programmation tout en me fabriquant une aide - je n'irais pas jusqu'à dire que c'était de la triche - pour les contrôles. Plus tard, j'ai tenté de plus gros projets : tout d'abord un éditeur qui permettait de dessiner plusieurs formes géométriques comme on le voulait, et ensuite de sauvegarder sa création. Mais aussi le projet qui me tenait le plus à cœur : une interface permettant de jouer au Puissance 4, ce célèbre jeu de société dans lequel il faut aligner quatre pions de sa couleur. Ce programme simpliste permettait uniquement de jouer chacun son tour et de choisir la colonne désirée avec les flèches, puis de valider pour poser un pion.

Puissance 4 en TI-Basic sur calculatrice TI-82 (ce n'est pas exactement le modèle de calculatrice que j'avais) :
Les touches 1 à 7 du clavier permettent de placer un pion dans la colonne correspondante. Le prochain symbole à être joué est indiqué en haut à droite.

En classe de terminale, j'ai eu une nouvelle calculatrice, plus moderne, et supportant le langage Python. Je me suis donc habitué à Python et sa syntaxe, et c'est à ce moment que j'ai compris que lorsque l'on a déjà appris un langage, il est plus simple d'en apprendre un autre. En effet, les concepts restent les mêmes, seuls les mots-clés changent, et c'est bien pour cela que l'on peut faire de "l'informatique théorique", qui n'utilise pas un langage précis. J'ai donc commencé à écrire des programmes en Python, mais toujours sur calculatrice, car cela permettait d'y avoir accès en cours et dans les transports. J'ai réalisé plusieurs jeux comme le classique *Snake*, et j'ai notamment refait le programme de Puissance 4 sur ma nouvelle calculatrice, en y ajoutant cette fois un système de détection de fin de partie, c'est-à-dire que le programme s'arrête quand il trouve quatre pions alignés.

Puissance 4 en Python sur calculatrice Numworks :
Les flèches gauche et droite permettent de choisir la colonne, la touche OK permet de placer un pion. La prochaine couleur à être jouée est indiquée en haut à gauche.

En entrant en classe prépa, j'ai donc commencé les cours d'informatique : de la programmation en Python, avec des travaux pratiques d'application sur ordinateur. C'est ainsi que, progressivement, j'ai abandonné les programmes sur calculatrice au profit de ceux sur ordinateur pour beaucoup de raisons, la plus évidente étant qu'écrire avec un clavier de calculatrice est fastidieux. On peut noter que cette transition, contrairement à celle entre calculatrices, n'impliquait pas de changement de langage, et donc était beaucoup plus fluide.

Nous avons déjà parlé de *Minecraft* et de son caractère "bac à sable" que je trouve si intéressant. Je vais maintenant expliquer comment je suis entré dans le monde du *modding* grâce à *Minecraft*, ce qui est, selon moi et dans mon cas, l'apogée du mélange de la pratique des jeux vidéo et du loisir de programmer. Pour cela, reprenons au début.

Mon approche dans le *modding*

Apparition d'une nouvelle pratique

Ainsi, quand j'étais plus jeune, j'ai joué à *Minecraft* plusieurs années de façon "normale" (i.e. sans utiliser de *mods*, car je n'avais pas les compétences pour utiliser ou faire ces modifications), jusqu'à ce que sorte une nouvelle fonctionnalité : la possibilité de mettre des commandes administrateurs - utilisées pour modifier le jeu "en temps réel", un peu à la manière de codes de triche - dans des fichiers (dits *mcfunction*) exécutés par le jeu de façon automatique. C'était le début des *data packs*, en français "packs de données", de simples dossiers modifiant le comportement du jeu, sans pour autant altérer son code originel parce que toutes les commandes sont déjà intégrées au jeu. Et même si la portée des *data packs*, qui ne sont donc pas des *mods*, est moins grande, dans le sens où l'on ne peut rien créer de zéro, la simplicité à installer et programmer combinée à la présence d'une communauté pleine d'inventivité pour mettre au point des techniques accomplissant telle ou telle modification, les *data packs* n'ont rien à envier aux *mods*.

Découverte et appropriation

J'y ai donc vu l'occasion parfaite de me lancer dans le *modding* : toutes les complications relatives aux *mods* purs et au fonctionnement du jeu avaient disparu. À la manière de l'apprentissage de n'importe quel langage, j'ai donc commencé par appréhender les mécanismes de base et les fonctionnalités essentielles. Cette phase est plus complexe que pour un "vrai" langage de programmation, car ils comportent souvent les mêmes outils : boucles *while*, *for*, pointeurs et tableaux… Une fois que l'on sait programmer dans un langage, mis à part quelques différences dans la façon de penser le code (ce sont les modèles tels que la programmation orientée objet ou la programmation impérative), il suffit donc d'apprendre la nouvelle syntaxe et les nouveaux mots-clés pour se débrouiller. Or la plupart de ces outils n'existent pas en *mcfunction*, il faut les construire à partir de ce que le jeu nous offre, c'est-à-dire entre autres des tableaux - les *scores* - dont la taille n'est pas fixée et qui sont indexés par les entités existantes dans le jeu, ou les *tags*, permettant de marquer les entités. En revanche, il existe des outils spécifiques au jeu : ce

sont les fichiers des *data packs* qui ne sont pas des *mcfunction*, mais des *json*, un format bien connu dans le monde de la programmation, et qui permet de définir de nombreux objets spécifiques à *Minecraft*, comme des prédicats, valant vrai ou faux selon la situation, des déclencheurs de fonctions et autres, facilitant ainsi certaines tâches.

Une reformulation mentale

Une fois les bases acquises, il est temps de se tourner vers un vrai projet. Il est parfois fastidieux de déplacer des animaux domestiques dans une partie normale. Un objet qui aurait la même fonction qu'une *Pokéball* de l'univers Pokémon - c'est-à-dire un objet permettant de "stocker" un animal - serait très utile. Après tout, c'est cette idée qui, en partie, m'a motivé à apprendre tout cela. Mais il est impossible d'implémenter directement cet objet, car il ne colle pas à l'esprit de *Minecraft*, c'est un objet bien trop avancé technologiquement, alors que dans le jeu, il s'agit de fabriquer des objets plutôt primitifs. En revanche, comme il existe déjà des objets magiques, par exemple des potions, je me suis dit que je pourrais peut-être "expliquer" cet objet *Pokéball* par la magie, ce qui collerait déjà beaucoup plus à l'esprit originel du jeu. Commençons par le renommer "Attrape-Âme", ce nom me semble plutôt logique pour un objet stockant un animal. Il faut également déterminer comment le joueur fabriquera cet objet, pour cela, nous pouvons choisir des objets existants du jeu qui possèdent déjà des propriétés magiques. Concernant l'apparence, donnons-lui un aspect qui mélange les apparences des objets nécessaires à sa fabrication.

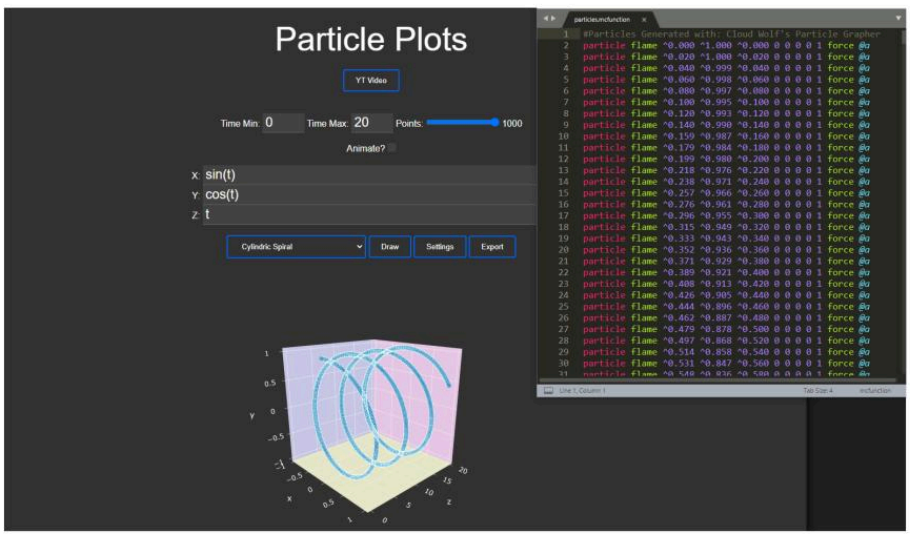
Textures (réalisées par un ami) de l'Attrape-Âme : à gauche, il est vide, et à droite, il contient un animal (l'étincelle est animée et tourne autour du centre).

Maintenant que nous avons la base théorique, l'idée précise du projet, il est temps de commencer à coder.

Motivations et techniques

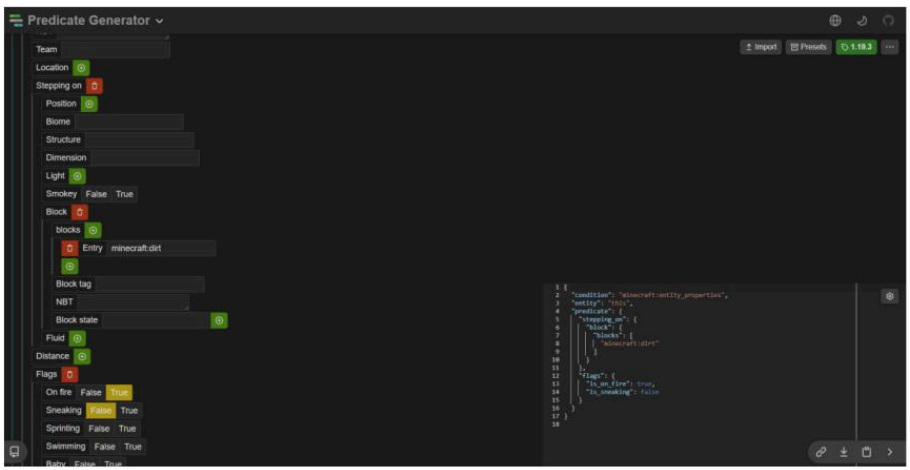
Pour tous mes autres projets, je suis globalement la même logique. Formalisons tout cela : tout d'abord, il peut y avoir différentes motivations à réaliser un nouveau projet : on peut vouloir combiner deux univers qui nous plaisent dans un seul, ce qui revient ici à injecter des éléments d'un univers dans *Minecraft*. Il s'agit donc de percevoir ces éléments sous un angle de vue qui les intègrent correctement au jeu de base. Cela peut aussi être l'occasion de donner vie à de nouvelles idées de fonctionnalités ou mécaniques de jeu, ou bien même de créer un jeu à priori nouveau, en combinant *data packs* et construction du décor, on peut réaliser un jeu utilisant *Minecraft* comme base. Tout ceci renforce la nature "bac à sable" de *Minecraft*. Enfin, quand ces ajouts ne sont pas tout simplement des défis de programmation, ils peuvent aussi être le moyen d'améliorer le jeu en ajoutant ou modifiant des fonctionnalités de manière *vanilla*, c'est-à-dire s'intégrant au jeu comme si elles étaient présentes par défaut. Vous connaissez maintenant les raisons qui peuvent me pousser à réaliser un projet, voyons désormais comment je procède de manière générale. Une fois l'idée trouvée, il faut commencer par une phase d'appropriation de l'idée ou du concept, pour en déterminer les subtilités, les interactions entre ses différents éléments et ceux existants de *Minecraft*, et remodeler sa forme, c'est-à-dire les apparences que vont prendre les ajouts, la façon dont ils peuvent être "expliqués", tout ceci dans plusieurs buts : gommer ce qu'on pense être des défauts, ajouter des fonctionnalités que l'on aurait voulu dans l'idée "existante" que l'on souhaite reprendre, mais aussi donner un style qui correspond à *Minecraft* (c'est essentiel dans le cas où l'on souhaite ajouter des fonctionnalités utilisables dans une partie normale, car sinon la frontière entre le jeu de base et les ajouts sera trop marquée, ce qui nuit à l'expérience du joueur). La dernière partie de cette phase, qui est aussi la plus importante, est ce que l'on appelle le *game balance* : il s'agit d'ajuster les variables numériques - par exemple le temps pour recharger une arme, le prix d'un achat, la quantité de matériaux pour fabriquer un objet - afin que l'expérience du joueur soit la meilleure possible, donc une progression ni trop difficile ou fastidieuse, ni trop facile ou rapide. Par exemple, dans le cas où l'on ajoute un objet, il ne faut pas que cet objet soit trop puissant, que son propriétaire soit trop avantagé par rapport à d'autres joueurs, que l'utilisation

d'un objet demande une contrepartie, ou bien qu'un objet soit impossible à fabriquer parce que nécessitant trop de ressources. Cet équilibrage nécessaire est souvent compliqué à trouver et demande plusieurs essais "en conditions réelles". Une fois que l'idée est globalement finalisée, la prochaine phase est la programmation. Dans le cas des *data packs*, il faut donc utiliser les outils mis en place par les développeurs, qui sont donc limités, et bien souvent, il faut ruser et appliquer le proverbe "reculer pour mieux sauter", dans le sens où la solution technique pour implémenter quelque chose n'a souvent aucun rapport avec le problème. Fort heureusement, la communauté de *data packs makers* comporte plusieurs personnes qui trouvent des moyens d'ajouter certaines fonctions complexes et qui partagent leurs travaux de façon à ce que les autres puissent utiliser ces ajouts "génériques" dans leurs propres *packs*, ces ajouts "génériques" peuvent être par exemple un système qui détecte les appuis de touche de clavier du joueur, ou bien un système permettant de réaliser des menus cliquables.



Site internet permettant d'afficher une courbe dans l'espace.
Cliquer sur le bouton Draw *génère le fichier à droite qui permet d'afficher la courbe dans le jeu grâce à un* data pack*.*
L'intérêt est ici d'automatiser le calcul des coordonnées xyz des points de la courbe.
https://cloudwolfyt.github.io/pages/gens/particle-plots.html

Un autre exemple est celui des générateurs : nous avons évoqué plus haut les fichiers *json*, qui, dans le cas des *data packs* permettent de formaliser des groupes d'objets ou des conditions. Ces fichiers suivent une syntaxe rigoureuse, et pour aider à leur écriture, certains membres de la communauté écrivent des générateurs sous forme de site internet dans lequel il suffit de cliquer sur les éléments que l'on désire pour obtenir le fichier correspondant (cela permet donc d'obtenir les fichiers que l'on souhaite sans risque d'erreurs, sauf dans certains cas comportant des subtilités qui n'ont pas de rapport avec le format *json*).



Site internet bien connu de la communauté, permettant de générer les fichiers json*, on complète à gauche et on obtient le fichier en bas à droite. Ici, on a un prédicat qui teste si le joueur est debout, enflammé, et marche sur un bloc de terre (cet exemple, qui n'a pas réellement de sens, montre que l'on peut tester beaucoup de choses de manière condensée). https://misode.github.io*

Ce sont en quelque sorte des boîtes à outils mises à disposition de tout le monde, et qui sont très utiles, car dans ce contexte comme dans les sciences en général, il est inutile de "réinventer la roue" à chaque fois que l'on veut faire quelque chose.

Échecs et entraide

Parfois, même en écumant Internet, il est impossible - ou du moins il semble impossible - d'implémenter certaines choses. Il faut alors revenir à la première phase, afin de remodeler l'idée-objectif pour modifier ou supprimer la fonctionnalité qui pose problème, ou bien même, dans le cas où la difficulté réside dans la correction d'un bug, transformer ce bug en fonctionnalité. Par exemple, lors de la programmation d'un sortilège censé retirer des points de vie pour un *data pack* sur le thème de la magie, j'ai fait face à un problème : la seule solution qui me paraissait pertinente à implémenter comportait un défaut, elle retirait bien des points de vie à la plupart des créatures, mais en donnait aux créatures de type "mort-vivant", par définition de l'outil que j'utilisais pour "simuler" le sortilège. Ne pouvant pas trouver de solution viable, j'ai donc décidé de "choisir" ce comportement, et j'ai créé le sortilège complémentaire, blessant les morts-vivants et soignant les autres créatures. Cette technique est d'ailleurs utilisée par les développeurs de *Minecraft* eux-mêmes : il est courant de voir, sur leur site de report de bugs, des bugs "corrigés" avec la mention *Works as intended*, autrement dit, *Fonctionne correctement*. Une fois que la base du travail est faite, il est nécessaire de récupérer des avis extérieurs de joueurs qui testent la création pour améliorer le *game balance* et les fonctionnalités[3]. On peut également faire appel à d'autres créateurs pour compléter son travail, aller plus vite, ou réussir plus de choses. Par exemple, on peut demander de l'aide à un ami dessinateur pour "retexturer" les objets qu'on ajoute, c'est-à-dire leur donner une nouvelle apparence. De manière générale, et comme dans tous les domaines, plus le projet est conséquent, plus il nécessite de contributeurs car les chances qu'il requiert des compétences très variées sont plus grandes.
Enfin, quand un projet semble abouti, que ce soit un *data pack* ou n'importe quel projet de programmation, il vient assez naturellement l'idée de le partager ou de le publier, afin que d'autres programmeurs puissent à leur tour s'en inspirer.

[3] Mathieu Cocq, "L'organisation et l'exploitation du travail des joueurs", *Réseaux*, N°213, 2019, pp 111-137

Conclusion

Nous voici au terme de cette découverte du *modding*. Nous avons vu que l'industrie du jeu vidéo a été construite au fil des révolutions technologiques de l'informatique, et qu'elle considère de plus en plus le joueur comme un créateur - à sa façon - du jeu. En prenant l'exemple de mon histoire avec le jeu vidéo et la programmation, j'ai tenté de souligner la réflexion nécessaire et la créativité exprimée dans la réalisation d'un *mod*, et pourquoi c'est, selon moi, une part non négligeable des jeux vidéo d'aujourd'hui. Tandis que j'écris ces lignes, le studio de développement de *Minecraft* prépare une mise à jour qui rajoutera plusieurs outils réclamés par la communauté et facilitant la création de *data packs*, ce qui prouve sa volonté de faire vivre cet aspect essentiel du jeu.

Références

COCQ Mathieu, "L'organisation et l'exploitation du travail des joueurs", *Réseaux*, N°213, 2019, pp 111-137

RECHSTEINER Alexander. "L'histoire des jeux vidéo" [en ligne]. Disponible sur : https://blog.nationalmuseum.ch/fr/2020/01/histoire-des-jeux-video/ [15/01/2023]

ROUET François, "La création dans l'industrie du jeu vidéo", *Culture Études*, N°1, 2009, pp 1-16

WIKIPÉDIA. "Histoire du jeu vidéo" [en ligne]. Disponible sur : https://fr.wikipedia.org/wiki/Histoire_du_jeu_vid%C3%A9o [15/01/2023]

Images de couverture : https://free4kwallpapers.com

Résumé

La programmation et les jeux vidéo sont deux sujets qui m'intéressent depuis longtemps. À travers cet écrit, je vous propose de découvrir l'évolution du jeu vidéo, puis d'explorer le lien qui existe entre ces deux pratiques grâce à ma propre expérience, notamment dans une forme de *modding*.

Ronan LEBAS