Machine Notionnelle et Pratiques de la Programmation: Mieux Apprendre avec le Développement Progressif

BOISVERT, Charles <http://orcid.org/0000-0002-3069-5726>

Machine Notionnelle et Pratiques de la Programmation: Mieux Apprendre avec le Développement Progressif

Charles Boisvert

Machine Notionnelle et Pratiques de la Programmation: Mieux Apprendre avec le Développement Progressif

Charles Boisvert[1]
(1) Department of Computing, Sheffield Hallan University, Sheffield S1 1WB, United Kingdom
c.boisvert@shu.ac.uk

RÉSUMÉ

Cet article définit les problèmes de développement progressif comme des problèmes dont les solutions partielles ou erronées sont utiles au progrès vers une solution complète. Appliquant cette définition à des exemples Scratch, il présente trois exercices (un jeu simple, un en trois dimensions et une simulation mécanique de bielle-manivelle) qui peuvent être traités comme des problèmes de développement progressifs, et montre comment cette présentation permet d'encourager les apprenants à développer des pratiques de programmation efficaces et une machine notionnelle — un modèle mental de l'interprétation du code — plus juste.

ABSTRACT

Notional Machine and Practice of Programming : Learning Better with Progressive Development

This paper defines progressive development problems as problems for which the partial or mistaken solutions help progress to a more complete one. Applying this definition to Scratch examples, it presents three exercises (a simple game, another in three dimensions, and a rod-crank mechanical simulation) that can be treated as progressive development problems, and shows how this presentation helps to encourage learners to develop better programming practices and improve their notional machine — mental model of code interpretation.

MOTS-CLÉS : développement progressif ; Scratch ; ressources éducatives ; machine notionnelle.

KEYWORDS: progressive development ; Scratch ; educational resources ; notional machine.

1 Apprendre à coder

L'enseignement de la programmation dès l'école primaire est grandement facilité aujourd'hui par de nombreux outils qui permettent de faire l'économie des difficultés de syntaxe et qui proposent de réifier certains concepts clés avec des objets virtuels ou même tangibles (Resnick et al. 2009). En particulier, la représentation du code par la métaphore de pièces de puzzle, qui ne permettent que les combinaisons syntaxiquement valides, a donné lieu à de multiples systèmes (Resnick et al. 2009, Harvey & Mönig 2010, Google 2019) utilisés aujourd'hui.

Mais ce progrès a de nombreuses limites. La revue menée par (Lodi et al. 2019) montre le besoin de développer l'enseignement de la programmation dans deux directions : en étendant la perception de la discipline, de la production de code à la résolution de problèmes associés ; et en développant chez l'apprenant la machine notionnelle, le modèle mental qui permet de suivre l'exécution d'un

Atelier « Apprendre la Pensée Informatique de la Maternelle à l'Université », dans le cadre de la conférence Environnements Informatiques pour l'Apprentissage Humain (EIAH), Fribourg, Suisse, 07 juin 2021, pages 57–67,

programme (Du Boulay 1986). Par exemple, les efforts de soutien menés par l'Université de Sheffield Hallam aux écoles locales (Adshead et al. 2015) se sont réduits après quelques années, les professeurs de primaire et de secondaire se satisfaisant d'un petit nombre d'activités qu'ils maîtrisent.

Il est clair que les enseignants ont relevé le défi de se former et de proposer rapidement cette nouvelle discipline, mais que les premières réponses ne peuvent pas suffire : le soutien doit se poursuivre dans le temps. Cet article propose, pour développer des apprentissages à l'école, de s'appuyer sur quelques notions clés de génie logiciel, et offre une piste pour les traduire dans l'enseignement.

2 Le code s'écrit pas à pas

La pratique de développer les programmes par étapes, vérifiant chacune avant de modifier le code est connue dans le génie logiciel. Comme le relève (Lodi et al. 2019), on retrouve ces pratiques dans les méthodologies agiles (Beck et al. n.d.) aujourd'hui communes chez les professionnels du développement ; cet usage de résultats exécutables incrémentaux est devenu central dans l'approche informatique des problèmes.

2.1 la Refactorisation

Par exemple, Daviaud et Revranche 2019, dans leur cahier de soutien, se servent de cette technique pour mieux faire comprendre le fonctionnement d'une boucle, comme présenté fig. 1. En proposant deux programmes aux résultats identiques, d'abord sans puis avec une boucle, les auteurs ne font pas que montrer l'avantage du second. Ils font usage d'une technique de génie logiciel, la refactorisation, qui consiste à réécrire un programme déjà fonctionnel pour le rendre plus facile à manipuler, sans pour autant en changer le résultat.

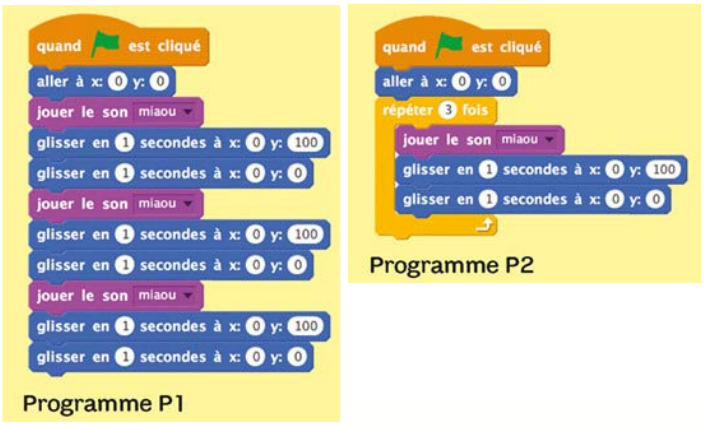

FIGURE 1 – Réécrire un programme : deux scripts aux effets identiques, l'un long, l'autre rendu plus concis par une boucle. Source : (Daviaud & Revranche 2019)

Cette technique permet aux développeurs de s'appuyer pendant la plus grande partie de leur projet sur des résultats intermédiaires exécutables, qui peuvent être évalués par comparaison avec une définition explicite du résultat fini et qui forment part du processus de résolution du problème.

2.2 Développement progressif : définition

On peut envisager que le développement de programmes par essai et erreurs aurait quelque chose de *naturel*, et donc va être *bon* dans nos méthodes d'enseignement. Mais l'exemple de la section 2.1 ci-dessus, n'est pas motivé par une erreur — ou en tout cas pas par une erreur de code ou d'exécution. Cette première idée ne suffit donc pas.

Pour mieux cerner comment cette notion se traduit dans une situation d'apprentissage, une définition plus claire, et restreinte, de ce qui fait qu'un problème de développement est progressif est proposée dans (Boisvert 2009). Il s'agit de problèmes où :

— La solution ne va pas de soi ;
— Les solutions partielles sont productives, c'est à dire qu'elles permettent un progrès vers une solution plus complète ;
— les solutions erronées, ou partiellement erronées, sont productives également.

Les problèmes de ce type proposés dans l'article d'origine sont des exercices bien connus dans l'enseignement de l'informatique au niveau supérieur, comme le tri, l'affichage d'un calendrier, ou le traitement de chaînes de caractères. Les enseignants du primaire et secondaire, ont besoin d'exemples de réalisations interactives et visuelles, plus motivantes pour de jeunes élèves, et adaptées à des environnements comme Scratch.

2.3 Déconstruire nos scripts Scratch

De nombreux manuels destinés aux élèves proposent des exemples Scratch qui sous-utilisent ce bénéfice d'une programmation pas-à-pas. Figure 2 montre, par exemple, un script extrait d'un jeu proposee par (Vorderman 2019). C'est un script assez commun, qui permet au joueur de contrôler les mouvements d'un lutin vers la droite ou la gauche avec le clavier.

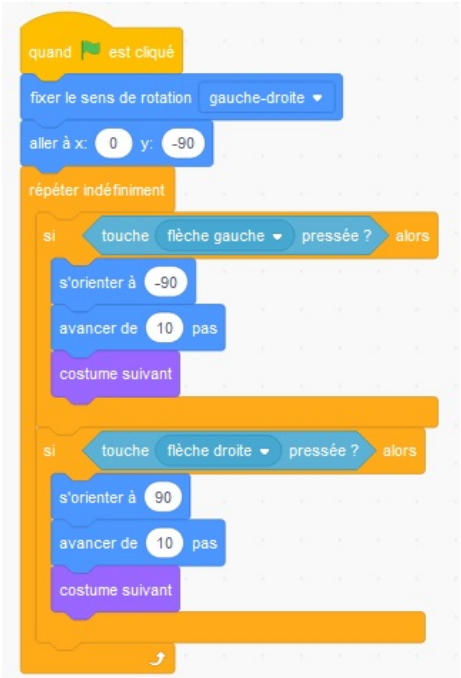

FIGURE 2 – Un script compliqué pour un mouvement simple. D'après (Vorderman 2019)

Le script utilise simultanément des effets visuels simples et un certain nombre de notions difficiles : sélections, événements, boucle, initialisation. Mais surtout il est difficile à utiliser autrement que d'une seule pièce. L'apprenant qui voudrait voir l'effet d'une partie du travail devrait choisir soigneusement quelle section ignorer pour vérifier un résultat. Une erreur de manipulation est aussi plus difficile à repérer pour la corriger. Enfin, si l'on voulait adapter le script, par exemple pour déplacer le personnage de haut en bas, ce point de départ encourage à créer peu de scripts faits de nombreux blocs : un phénomène de 'code spaghetti' bien connu des professionnels. Mêmes plus colorés, les spaghettis de Scratch n'en deviennent pas moins rapidement illisibles.

Une alternative à l'exemple ci-dessus consiste au contraire à refactoriser le code en séparant chaque résultat souhaité en scripts courts à l'objectif clair. Cette version éclatée est présentée figure 3. La version refactorisée permet d'exécuter chaque script sans devoir assembler les autres. On peut aussi plus facilement cibler une erreur de copie ou de conception. Enfin, la série de scripts est plus facile à étendre ou à modifier.

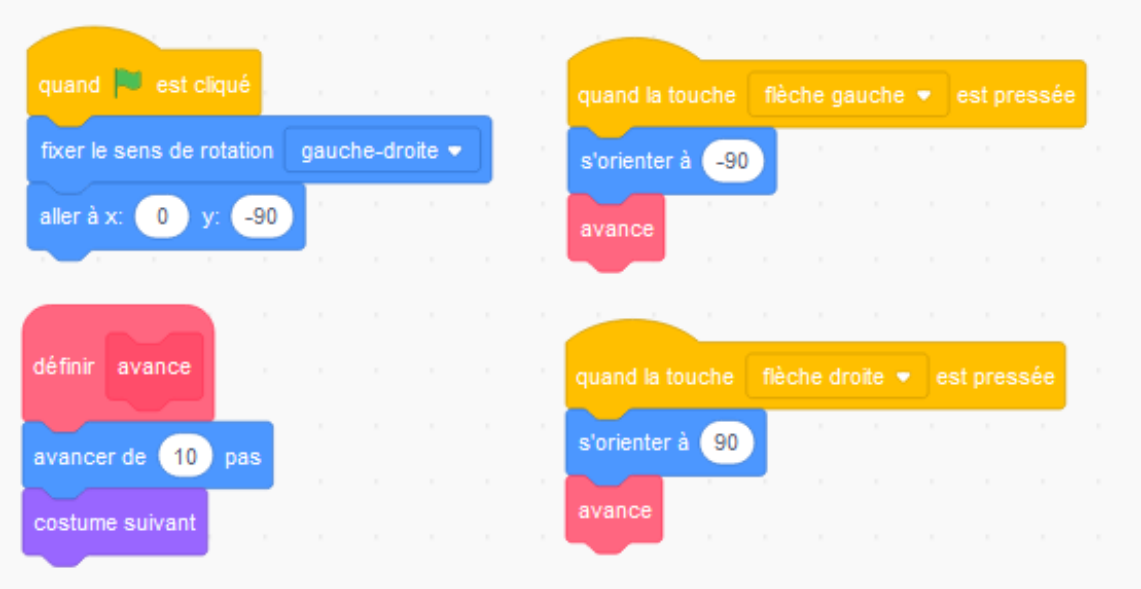

FIGURE 3 – Le même résultat que figure 2 en traitant chaque objectif (initialisation, déplacement dans chaque direction, animation du mouvement) par un script séparé.

On pourrait s'inquiéter de ce que cette seconde version fait appel à des notions complexes de parallélisme et d'événements. Les exemples de cet article utilisent en effet les deux. Cependant, Scratch facilite la compréhension, en permettant de réifier ces concepts. Ces notions sont plus complexes dans un contexte plus large — où le langage de programmation, la diversité des systèmes, la persistance d'anciennes approches, l'exigence de garantir la sécurité et la stabilité, imposent une compréhension détaillée et fine des notions et une pratique plus exigeante qui rend souvent le code confus.

Mitch Resnick raconte qu'avec son équipe, à la conception de Scratch, ils avaient 'peur de tout ce qui pourrait effrayer un enfant de huit ans' [1]. L'équipe Scratch a recherché des moyens de représenter le parallélisme ou les événements de telle façon que leur usage ne fasse pas appel aux notions abstraites nécessaires pour les comprendre en détail. Dès lors, on peut se servir de ces outils en pratique : de la même façon qu'on n'hésite pas à lancer un ballon à des enfants, même s'ils ne sont pas encore en

1. Communication orale : discours d'accueil, Scratch Bordeaux 2017

mesure de saisir tous les détails de la gravitation universelle, il ne faut pas craindre de leur donner deux script qui fonctionnent en même temps, même s'ils ne sont pas encore en mesure de saisir tous les détails de l'exécution parallèle. Peut-être même qu'au contraire, de même que l'enfant qui a joué au ballon pourra un jour se servir de cette expérience pour mieux conceptualiser le calcul balistique, celui ou celle qui aura vu Scratch exécuter deux scripts simultanément pourra s'appuyer sur cette expérience pour intégrer les concepts du parallélisme.

Pour mieux appréhender cette pratique, considérons des exemples plus complets choisis pour associer problème à résoudre et usage de code imparfait comme moyen de résolution.

3 Trois problèmes progressifs en Scratch

De nombreux exemples en Scratch peuvent être adaptés au développement progressif. Les trois cas ci-dessous illustrent l'usage que l'on peut faire des fonctionnalités de Scratch pour présenter un développement dont les étapes sont productives et testables.

3.1 Poser la fusée

L'atterrissage d'une fusée est bien connu comme jeu et comme exercice de programmation. La version présentée ici vient d'une collection de ressources créée par Martin Quinson (2014). Le problème en Scratch, et une solution qui fait usage du parallélisme sont illustrés fig. 4.

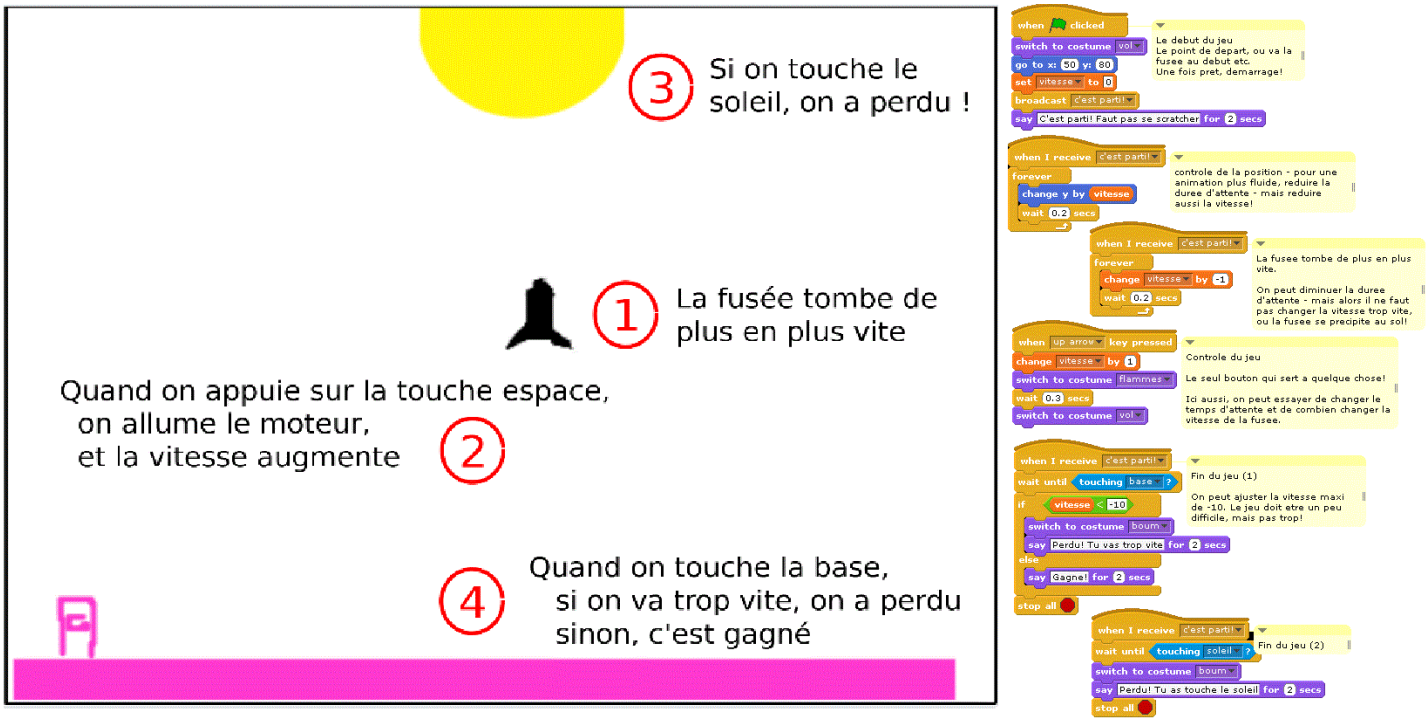

FIGURE 4 – Poser la fusée : le problème (à gauche) et une solution utilisant le parallélisme de Scratch (à droite)

On y retrouve plusieurs techniques qui aident au développement et à la compréhension du programme :

— Plusieurs scripts courts plutôt qu'un long. Chaque script peut-être écrit et testé séparément, permettant de vérifier et de réfléchir sur des résultats partiels : *les solutions partielles sont productives.*
— Décomposition des problèmes. Chaque script accomplit une seule fonction dans le jeu (de haut en bas, fig. 4 : démarrage du jeu, mouvement de la fusée, accélération de sa chute, contrôle par le joueur, et deux conditions de fin).
— Un schéma (*design pattern*) réutilisable. L'initialisation du jeu est confiée à un script unique qui établit les conditions initiales puis appelle le coeur du jeu.

Trop d'exemples proposent, comme dans le cas de la figure 2 (Vorderman 2019), de copier en entier un long script avant de voir un résultat. Cette approche permet d'introduire les concepts un par un et de vérifier leur fonctionnement par étapes. L'exercice complet (Quinson 2014) présente chaque étape du développement et insiste sur la possibilité d'exécuter le programme incomplet et de personnaliser le jeu au fur et à mesure. Elle s'étend facilement à de nombreux jeux et animations sur Scratch.

Le test d'un script en Scratch peut être l'occasion de mettre en pratique une technique de génie logiciel. Même pour de jeunes apprenants, on peut retrouver :

— *Une définition claire de 'fini'* : par exemple, il faut pouvoir exprimer à l'avance que 'si ça marche, la fusée va...'
— *Des tests multiples* : Un essai unique ne suffit généralement pas ; est ce que la fusée peut monter et descendre ? Qu'est-ce qui se passe si elle va plus vite ?
— *Une prise en compte des cas exceptionnels* : que fait la fusée au départ de l'animation, à la fin ?

3.2 Un peu de 3D

Le second exemple est tiré des ressources développées pour l'enseignement secondaire à l'Université de Sheffield Hallam (Adshead et al. 2015). L'une d'elles est une série de jeux du même type : le joueur voit arriver vers lui des objets (ennemis à abattre, obstacles, éléments de décor). Leur mouvement et la perspective créent l'illusion d'un parcours de jeu.

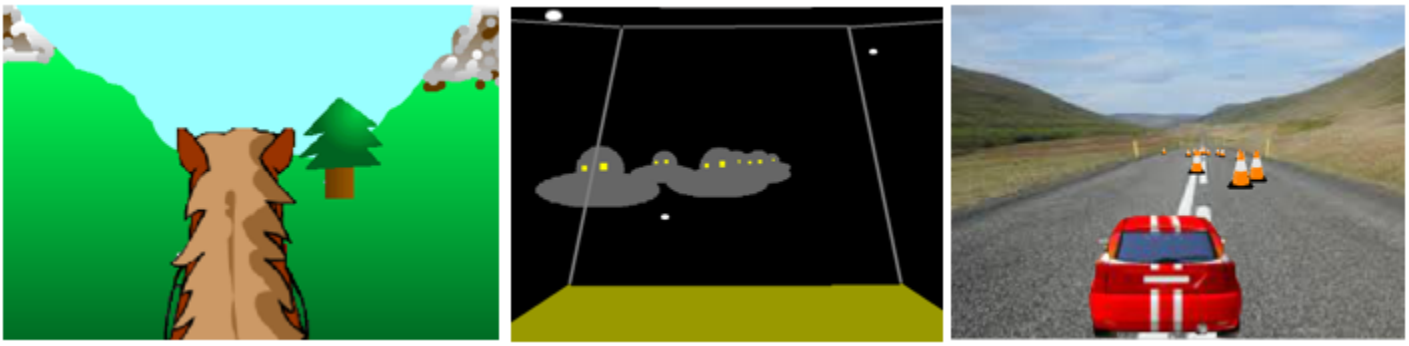

FIGURE 5 – Une série de jeux utilisant la perspective

Une spécificité de cet exemple est que des apprenants jeunes ne maîtrisent pas les calculs nécessaires. Le matériel pédagogique [2] propose donc un bloc déjà prêt, présenté figure 6, pour projeter les trois

2. https ://tinyURL.com/3DScratch

coordonnées spatiales en deux coordonnées apparentes, et explique son usage sans entrer dans les détails mathématiques.

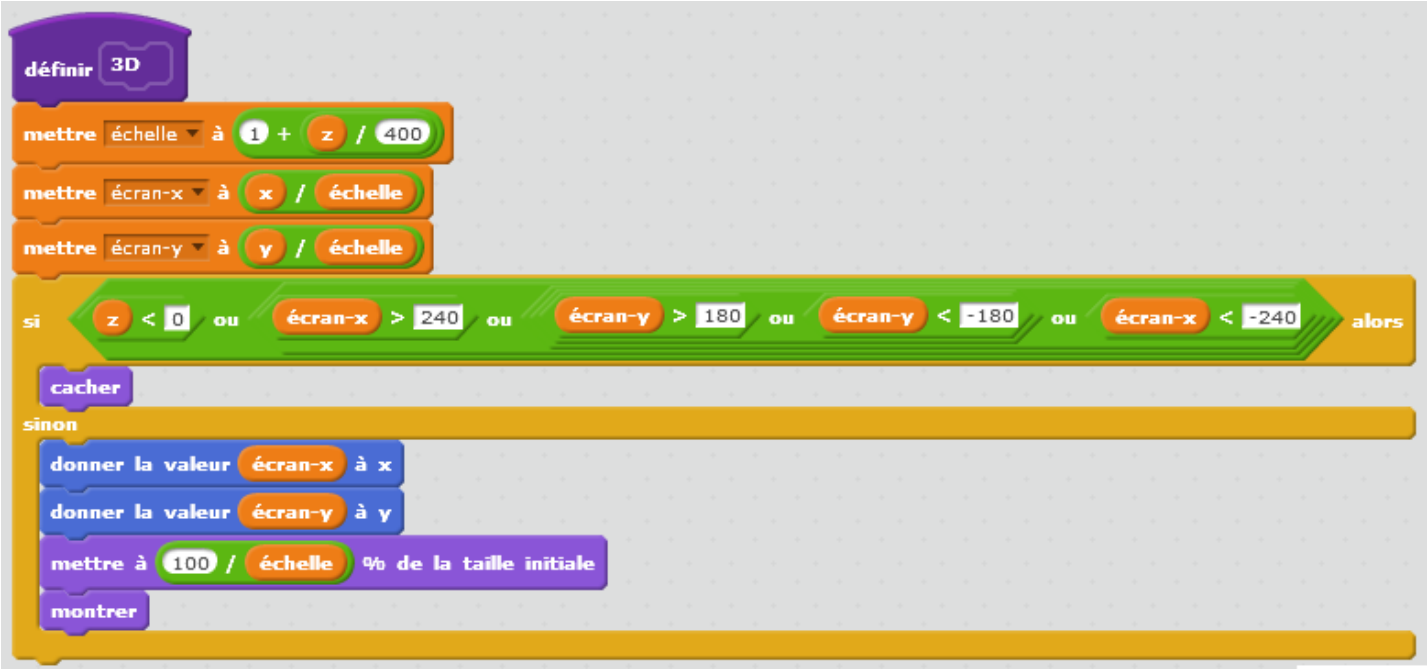

FIGURE 6 – Calcul des coordonnées X, Y et la taille apparentes d'un objet à l'écran à partir de variables de position spatiales x, y, z

Hormis ce bloc de calcul, les techniques précédentes, et en particulier le parallélisme de Scratch, permettent de définir des problèmes distincts et testables séparément : le mouvement des objets vers le joueur, les déplacements du joueur pour les atteindre ou les éviter, la gestion des collisions, forment chacun des fonctions distinctes.

Le matériel propose aussi aux apprenants de prendre le contrôle de leur jeu, de plusieurs façons :

— *Costumes à l'avance* : plusieurs lutins et fonds sont proposés, donnant des thèmes différents (course automobile, espace, balade à cheval). L'objectif est double : éviter que les apprenants ne passent trop longtemps à éditer les graphiques, et donner des options dès le départ. Les costumes sont proposés, avec le bloc de calcul 3D, dans un projet Scratch à modifier.
— *Offrir des options de jeu* : proposer des solutions alternatives au cours du développement. De nombreux exercices proposent une réalisation complète avant d'inviter l'apprenant à la modifier. La réalisation progressive et les tests d'exécution en cours de développement permettent de proposer des variantes alors que le jeu n'est pas terminé.
— *Bien choisir le bloc fourni* : le choix de ce que fait un bloc préparé d'avance demande beaucoup de soin. Il s'agit de faciliter une compréhension du problème en circonscrivant une opération et en la nommant. Ce choix limite et oriente les possibilités des apprenants, autant qu'il facilite certaines créations.

3.3 Un mécanisme bielle-manivelle [3]

Ce troisième exemple n'est pas un jeu mais une animation, montrant le mouvement d'un mécanisme bielle-manivelle. Le résultat est présenté figure 7.

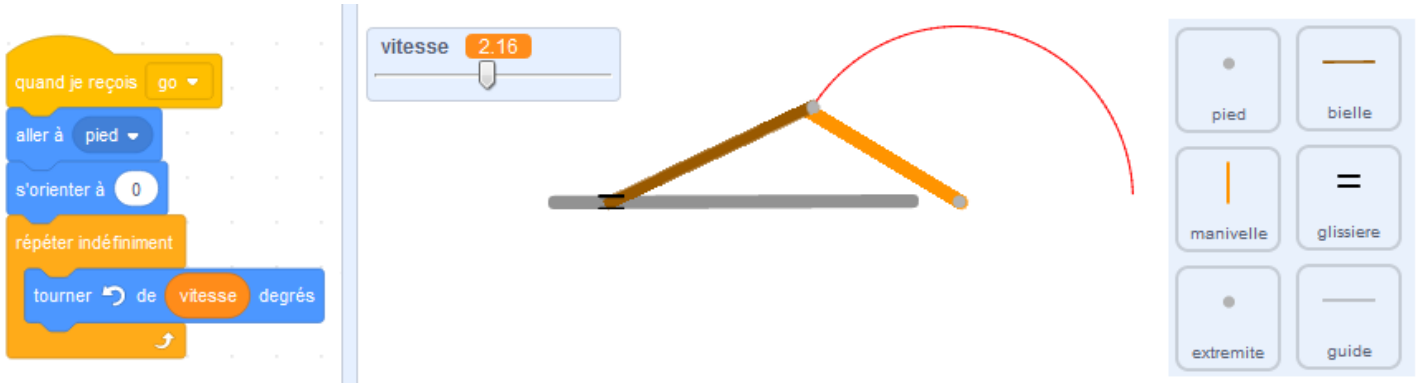

FIGURE 7 – Mécanisme bielle-manivelle. La décomposition permet de traiter le mouvement en coordonnant une série de scripts très simples, comme celui (à gauche) de la manivelle

Comme les deux précédents, le problème s'appuie sur des scripts courts qui font usage du parallélisme, et chaque sous-problème est traité par un script testable indépendamment des autres. Cependant, l'exécution en parallèle n'est pas aussi visible : le mouvement de chaque lutin est contrôlé par un unique script. La particularité de l'exercice se trouve dans l'association très claire entre les parties du mécanisme et la gestion de chaque élément du mouvement. Six lutins (à droite figure 7) modélisent le mécanisme. Deux d'entre eux (le pied et le guide horizontal) sont fixes. Le mouvement des quatre autres est simple : la manivelle et son extrémité tournent autour du pied ; la bielle suit l'extrémité de la manivelle et reste orientée vers la glissière ; la glissière enfin est maintenue à une distance toujours identique par la bielle.

Le positionnement d'un objet qui tourne autour d'un autre demande un calcul trigonométrique, pour lequel un bloc est fourni à l'avance, indiqué figure 8. Ce bloc utilise une autre notion abstraite qui peut inquiéter un enseignant mais qui, dans un contexte pratique d'usage, ne gêne pas les apprenants : le nom du bloc qui sert de point de référence est donné en paramètre [4].

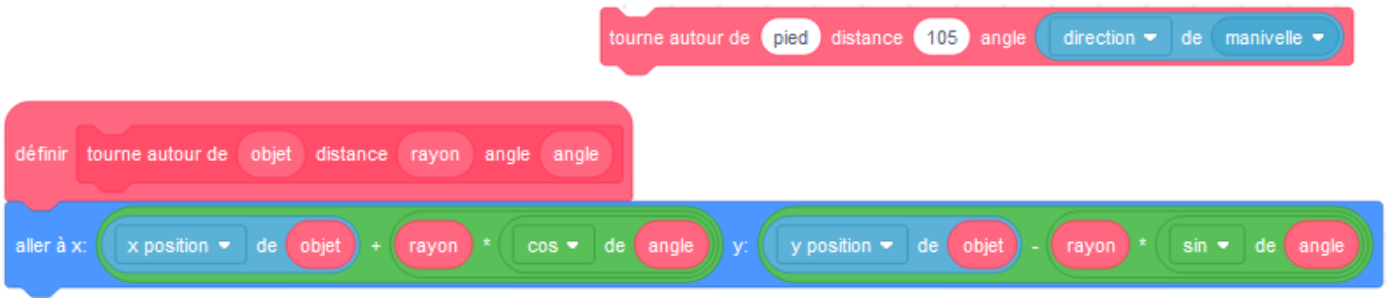

FIGURE 8 – Un bloc pour positionner un lutin autour d'un objet de référence, dont le nom est donné en paramètre.

3. https ://scratch.mit.edu/projects/87248378
4. Scratch interprète ce paramètre correctement mais traduit le bloc incomplètement quand on change la langue de l'interface

Avec des apprenants plus jeunes, il est préférable de fournir à l'avance les lutins décomposant le mécanisme. L'exercice consiste alors à trouver chaque mouvement. Comme dans les exemples précédents, on peut commencer par définir le résultat attendu, afin de tester les scripts, à mesure de leur développement, par comparaison à cette définition.

La recherche d'une solution donne lieu à toutes sortes de résultats incorrects, mais utiles à la compréhension du problème et du code, faisant du code erroné un moyen de réflexion. Seymour Papert illustre ce point magnifiquement dans son travail sur Logo (Papert 1980), reproduit figure 9 : les premières tentatives sont mauvaises, mais en voir le résultat nourrit la réflexion et un enfant ne trouverait peut-être pas le résultat sans ces 'erreurs' qui sont autant d'indications.

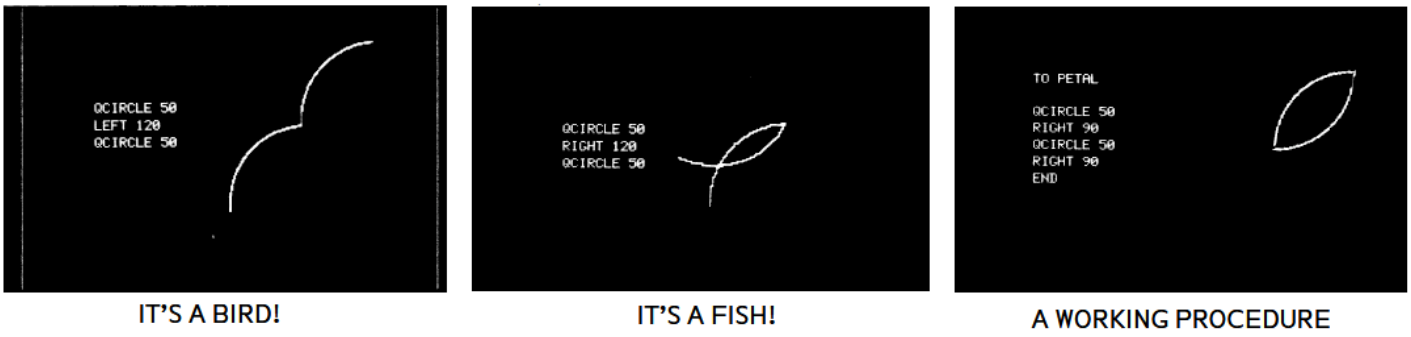

FIGURE 9 – 'C'est un oiseau !', 'C'est un poisson !' : le processus de débogage selon Papert (1980)

Le mouvement de la glissière est le plus difficile. Une solution consiste à comparer la distance de la glissière à l'extrémité de la manivelle pour la positionner : cette option présente l'avantage de s'accorder à un modèle mental du mécanisme — la bielle repousse ou tire la glissière.

Figure 10 présente une seconde option, qui peut être donnée toute prête. La solution part du même principe, mais calcule le défaut de distance pour mieux positionner la glissière.

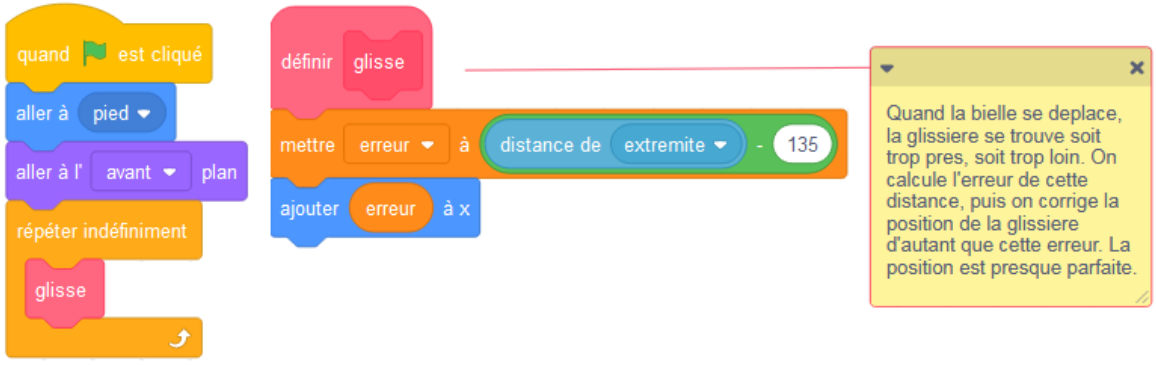

FIGURE 10 – Une manière de calculer la position de l'extrémité de la bielle.

.

Pour des apprenants plus âgés, cet exercice peut être l'occasion de rechercher une décomposition du problème. Le mécanisme bien visible soutient les apprenants dans leur recherche de solution, mais la résolution exige une étape de réflexion préalable, comme le recommande (Chevalier et al. 2020). En particulier, un bon choix du centre de chaque lutin simplifie la programmation de son mouvement, et la solution complète du mouvement de la glissière fait appel à des connaissances mathématiques (théorème de Pythagore).

4 Conclusion : Machine Notionnelle et pratiques de la programmation

La revue de (Lodi et al. 2019) fait valoir que la machine notionnelle — le modèle mental permettant de prévoir le comportement de leur système (Du Boulay 1986) — formée par de nombreux jeunes apprenants est largement simplifiée, s'appuyant souvent sur une personnification de l'interpréteur, qui plus est souvent représenté par un lutin animé. Pour l'auteur, les éducateurs et les apprenants ont besoin de soutien sur ce point, ainsi que de soutien pour appliquer des schémas et des pratiques de développement et de résolution de problèmes. Plutôt que de nouveaux outils technologiques, les exemples proposés ici tentent de répondre par des ressources utilisables dans le cadre technologique existant. En effet, l'usage pédagogique a autant, et quelquefois plus d'importance que l'outil (Hundhausen et al. 2002).

Dans le cas de Scratch, le développement de code spaghetti est une sous-utilisation fréquente du système. Les exemples proposés ici se servent du parallélisme pour à la fois développer une machine notionnelle plus complète, pour introduire une forme simple de décomposition, et pour dissocier les comportements (décrits par les scripts) des agents (décrits par les lutins). Simultanément, ils permettent de mettre en application certaines pratiques établies du développement logiciel : les méthodes de test pour guider le développement, la séparation des problèmes, et l'analyse du problème pour identifier les composants à traiter.

Références

Adshead, D., Boisvert, C., Love, D. & Spencer, P. (2015), Changing culture : Educating the next computer scientists, *in* 'Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education', ACM, pp. 33–38.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunnigham, W. & Fowler, M. (n.d.), 'Manifesto for Agile Software Development'.
URL: *https ://agilemanifesto.org/*

Boisvert, C. R. (2009), A visualisation tool for the programming process, *in* 'Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education', ITiCSE '09, Association for Computing Machinery, New York, NY, USA, pp. 328–332.
URL: *https ://doi.org/10.1145/1562877.1562976*

Chevalier, M., Giang, C., Piatti, A. & Mondada, F. (2020), 'Fostering computational thinking through educational robotics : a model for creative computational problem solving', *International Journal of STEM Education* **7**(1), 39.
URL: *https ://doi.org/10.1186/s40594-020-00238-z*

Daviaud, D. & Revranche, B. (2019), *Mini Chouette Programmer avec Scratch 5e/4e/3e : cahier de soutien en maths*, Hatier.

Du Boulay, B. (1986), 'Some difficulties of learning to program', *Journal of Educational Computing Research* **2**(1), 57–73. Publisher : SAGE Publications Sage CA : Los Angeles, CA.

Google (2019), 'Google blockly'.
URL: *https ://developers.google.com/blockly*

Harvey, B. & Mönig, J. (2010), 'Bringing "no ceiling" to scratch : Can one language serve kids and computer scientists ?', *Proc. Constructionism* pp. 1–10.

Hundhausen, C. D., Douglas, S. A. & Stasko, J. T. (2002), 'A Meta-Study of Algorithm Visualization Effectiveness', *Journal of Visual Languages & Computing* **13**(3), 259–290.
URL: *https ://www.sciencedirect.com/science/article/pii/S1045926X02902375*

Lodi, M., Malchiodi, D., Monga, M., Morpurgo, A. & Spieler, B. (2019), 'Constructionist Attempts at Supporting the Learning of Computer Programming : A Survey', *Olympiads in Informatics* **13**, 99–121.
URL: *https ://ioinformatics.org/page/ioi-journal-index/44volume13*

Papert, S. (1980), *Mindstorms : Children, computers, and powerful ideas*, Basic Books, Inc.

Quinson, M. (2014), 'quatre jeux · coding4kids'.
URL: *https ://github.com/mquinson/coding4kids*

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. & others (2009), 'Scratch : programming for all', *Communications of the ACM* **52**(11), 60–67. Publisher : ACM.

Vorderman, C. (2019), *Computer Coding for Kids : A unique step-by-step visual guide, from binary code to building games*, DK Children.