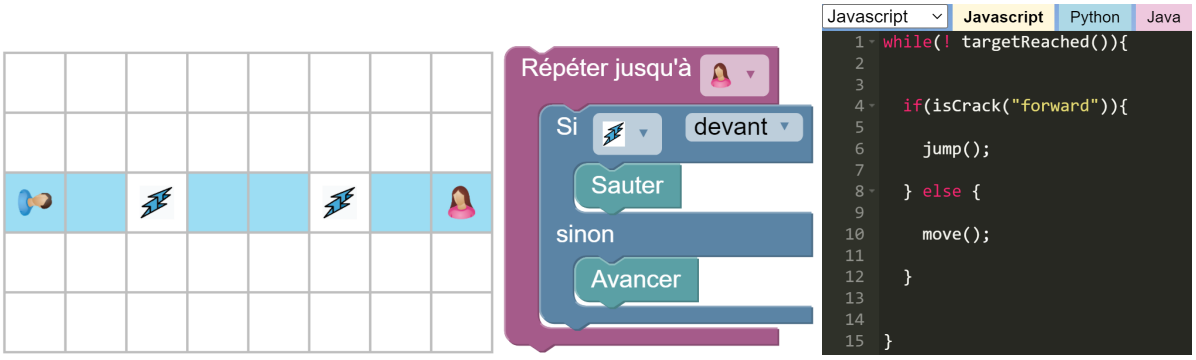
Sauvez Julie !

Développement d'un environnement d'apprentissage polymorphe de programmation visuelle et littérale



Travail de diplôme

Laurent Bardy

août 2023

Superviseur :

Prof. Dr. Jacques Pasquier
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Formation GymInf
Département d'informatique
Université de Fribourg
(Suisse)

GymInf
Informatikausbildung für Gymnasiallehrkräfte
Formation en informatique destinée aux enseignant-e-s au gymnase

"Rien dans les langages humains, aucune traduction de la pensée faite à l'aide des couleurs, des marbres, des mots ou des sons, ne saurait rendre le nerf, la vérité, le fini, la soudaineté du sentiment dans l'âme !"

Honoré de Balzac

Résumé

L'enseignement de l'algorithmique et de la programmation à des débutants pose nécessairement la question du moyen d'apprentissage. Celle-ci focalise souvent sur le choix du langage de programmation de haut niveau le plus approprié. La réponse à cette question est cependant plus complexe qu'il n'y paraît, dépendant notamment du stade de développement cognitif de l'apprenant. La théorie de l'apprentissage du psychologue Jean Piaget fournit des éléments d'analyse afin de formuler une réponse à cette question. Selon Piaget, l'enfant apprend et construit – en traversant différents stades d'apprentissage à travers les âges – son savoir par la pratique, en manipulant des objets physiques et abstraits de son environnement. Le choix du moyen d'apprentissage approprié dépend donc de l'âge de l'enfant, déterminant le stade auquel celui-ci se trouve. L'émergence récente des langages visuels ajoute une nouvelle couche d'abstraction se superposant à celles des langages littéraux de programmation de haut niveau, rendant l'apprentissage de la programmation accessible à un stade plus précoce, mais pouvant également faciliter, dans un premier temps, cet apprentissage chez une personne se trouvant déjà à un stade plus avancé. La littérature montre qu'un usage trop important et tous azimuts du numérique à l'école tend à prétériter les apprentissages fondamentaux, ce qui laisse à penser qu'un apprentissage de la programmation aux premiers stades de développement cognitif peut se révéler contre-productif. Si l'âge de l'enfant est un facteur déterminant pour le choix d'un moyen d'apprentissage adéquat, il est est également déterminant quant à l'objet d'apprentissage que permet de manipuler l'outil retenu, à l'instar d'une tortue dessinant sur un écran, d'un robot ou d'une page web par exemple. Plus l'objet manipulé est de nature complexe, plus la prise en main du moyen d'apprentissage nécessite un stade de développement cognitif élevé. Si les langages de programmation visuels offrent la possibilité de découvrir la programmation au stade d'apprentissage plus précoce des opérations concrètes, l'apprentissage d'un langage littéral sera plus adéquat au stade supérieur des opérations abstraites. Néanmoins, si les moyens d'apprentissage de la programmation sont aujourd'hui nombreux, peu d'entre-eux sont de nature polymorphe, offrant la possibilité, pour un même objet d'apprentissage, d'écrire le code d'un programme dans un langage visuel tout comme dans un langage littéral, encore moins dans plusieurs langages de haut niveau différents, ce qui facilite la transition d'un stade d'apprentissage à un autre tout en permettant une généralisation des notions fondamentales de programmation, existant dans tout langage de programmation de haut niveau, qu'il soit visuel ou littéral. *Sauvez Julie* est une application web expérimentale offrant un moyen d'apprentissage polymorphe portant sur la manipulation d'un objet d'apprentissage simple, à savoir une figurine sur un plateau de jeu quadrillé, représentant la surface d'un glacier sur lequel se déplace un guide de haute montagne devant aller secourir une touriste égarée. L'application repose sur différentes technologies (*HTML*, *CSS*, *Javascript*) et autant de bibliothèques et framework (*Processing* et *p5js*, *Blockly*, *JQuery*, *Ace Editor*), ses principales composants étant un éditeur de code visuel, un éditeur de code littéral, des convertisseurs de code visuel en code littéral, un moteur d'animation séquencé, un éditeur d'exercices et des tutoriaux pouvant être ajoutés aux exercices. Au joueur d'écrire le code visuel ou littéral d'un programme générant un itinéraire que le guide va suivre en toute sécurité pour sauver la touriste. Le code visuel du programme peut être converti automatiquement en différents langages littéraux de haut niveau mais le joueur a également la possibilité d'écrire par lui-même le code littéral en Javascript. L'expérimentation de cette application avec un enfant faisant la transition du

langage visuel au langage littéral montre une stratégie d'apprentissage par la pratique à laquelle ne fait le plus souvent pas appel un enseignement conventionnel, alors même que les informaticiens expérimentés procèdent également régulièrement par analogie et adaptation de code afin de passer d'un langage de programmation à un autre. Ces observations posent la question de la pertinence de l'approche déductive théorique par rapport à une approche inductive pratique, d'un apprentissage basé sur l'écriture ex-nihilo de code sources par contraste avec celui plus pragmatique se fondant sur l'observation, la compréhension et l'adaptation de codes existant. *Sauvez Julie* est un prototype expérimental de moyen d'apprentissage de la programmation disponible en *opensource* et dont le développement peut se poursuivre par toute personne intéressée.

Préambule

Remerciements

Mes remerciements vont tout d'abord à ma famille, pour son soutien tout au long de la réalisation de ce travail, mais également au cours des deux ans de formation GymInf ainsi que durant les deux années l'ayant précédée, consacrées à la mise en oeuvre du nouveau cours d'introduction à l'informatique au gymnase. Merci à Maude mon épouse pour sa patience durant les nombreuses et longues soirées ainsi que les samedis dédiés à cette formation et ces cours, y compris durant la période Covid. Merci également à mes enfants Manon et Damien pour leur enthousiasme intarissable à tester *Sauvez Julie* ! Qu'ils gardent en eux cette joie de la découverte et cette capacité d'émerveillement tout au long de leur vie. Mes remerciements vont également à Jacques Pasquier, professeur au Département d'informatique de l'Université de Fribourg, pour ses conseils toujours très pertinents et le suivi de ce troisième travail effectué sous sa direction en trente ans. Le premier travail, en programmation, avait été réalisé alors que j'étais encore jeune étudiant dans les années '90, le second une quinzaine d'années plus tard sur les bases de données lors de la formation DAS OCI s'adressant aux premiers enseignants d'informatique au gymnase appelés à y donner cette discipline en option et le troisième achevant la formation GymInf aboutissant à un ultime diplôme universitaire en informatique pour les enseignants de gymnase. Que Jacques garde durant sa retraite prochaine, à près de 70 ans, cet âme d'enfant qui l'a animé tout au long de sa carrière. Merci également à mes élèves ayant testé *Sauvez Julie* durant les premières années du cours d'introduction à l'informatique. Merci enfin à la Direction de la formation et des affaires culturelles (DFAC) du Canton de Fribourg pour avoir permis et soutenu cette formation.

Notations et conventions

A moins que cela ne se soit explicitement précisé dans le texte, l'usage du masculin ou du féminin pour des mots désignant des personnes fait référence aussi bien à des femmes qu'à des hommes.

Accès en ligne

L'application *Sauvez Julie !* est une application web accessible à l'adresse suivante :

https://apps.thike.ch/savejulie

La version électronique de ce travail ainsi que le code source de l'application (Github) sont disponibles via les liens que l'on trouve dans l'application web elle-même (voir liste des topos).

Table des matières

1. Introduction

"Quel langage de programmation va-t-on utiliser ?" Telle fut la question posée après un quart d'heure de discussion lors de la première réunion d'un groupe de travail ayant reçu pour mandat de rédiger un plan d'études d'un cours d'introduction à l'informatique au gymnase. La question revint de manière récurrente tout au long de la vingtaine de séances qui suivit. Est-ce que le langage A convient mieux que le langage B pour apprendre à programmer ? Avec en arrière-plan une question plus générale : le langage B est-il mieux que le langage A ? Question incendiaire pouvant allumer le feu d'un débat doctrinal très émotionnel chez les informaticiens. "Le meilleur langage ? Celui que j'utilise, assurément !" Impossible de l'éteindre en se recentrant sur l'objet premier de la discussion, à savoir les objectifs du plan d'études et donc les compétences spécifiques en informatique à développer chez les gymnasiens. Avant de discuter des moyens, encore faut-il au préalable définir les objectifs à atteindre pour déterminer ensuite les moyens en adéquation avec ceux-ci.

Réponse fut donnée à cet argument que le moyen retenu a une incidence sur les objectifs pouvant être atteints. Et donc qu'il est primordial de d'abord choisir le moyen avant de définir les objectifs. Ce qui, a priori, est également cohérent. On ne traverse par un océan avec un matelas pneumatique. Sauf qu'en sens inverse, personne ne choisira un matelas pneumatique s'il s'est donné comme objectif une telle traversée… Et qu'en l'occurrence les objectifs et concepts spécifiques du volet 'programmation' d'un cours d'introduction à l'informatique n'ont rien de révolutionnaire, n'ayant pas changé depuis des décennies : instruction, séquence d'instructions, variable, expression, structures de contrôle, fonction, bloc d'instructions et imbrication, etc. Et que tous les langages de programmation de haut niveau permettent de mettre en œuvre ces concepts, moyennant quelques différences de déclinaison syntaxique spécifiques. Certes, des langages peuvent se distinguer par leur degré de verbosité, la sobriété pouvant quelque peu faciliter l'apprentissage des fondamentaux. Ce ne sont en revanche pas des différences syntaxiques mineures qui jouent ici un rôle déterminant : délimiter un bloc d'instructions avec des accolades ou des indentations ne change guère la difficulté d'apprentissage de la notion.

Question subsidiaire à celle du choix du langage de programmation, mettant encore plus d'huile sur le feu : faut-il utiliser un langage de programmation unique imposé dans tous les cours ? Et ce dans l'intérêt des élèves changeant d'enseignant entre deux années de cours. Si la question de la stabilité se pose effectivement dans un même cours d'introduction tout au long d'une année – où l'on peut difficilement concevoir que le jonglage entre deux langages ne soit pas une difficulté inutile, si par exemple un éventuel changement s'effectue entre l'apprentissage des structures de répétition et celles de branchement – elle s'articule différemment concernant la transition d'une année à une autre. A la fin d'un premier cours, certaines bases ont été posées, des concepts ont été assimilés. On s'accommode dès lors plus aisément de quelques changements syntaxiques mineurs dans le cas de langages de programmation proches, moyennant une séance initiale de révision des concepts étudiés l'année précédente. Mieux, ce faisant on développe une aptitude de transcodage permettant de généraliser les notions apprises, en prenant conscience qu'elles ne sont pas spécifiques à un langage donné mais bien de nature universelle. Une telle généralisation offre également la possibilité de focaliser sur les fondamentaux en comparant deux codes similaires, de réaliser qu'il n'y a finalement que peu de différences quant à leur expression dans deux

langages différents (figure 1.1.), ces derniers n'étant au final que de simples outils et non une finalité en soi.

Si une telle transposition se révèle insurmontable ou particulièrement difficile, il y a lieu de s'interroger sur l'atteinte des objectifs fondamentaux lors du premier cours, avec des concepts qui n'auraient pas vraiment été assimilés.

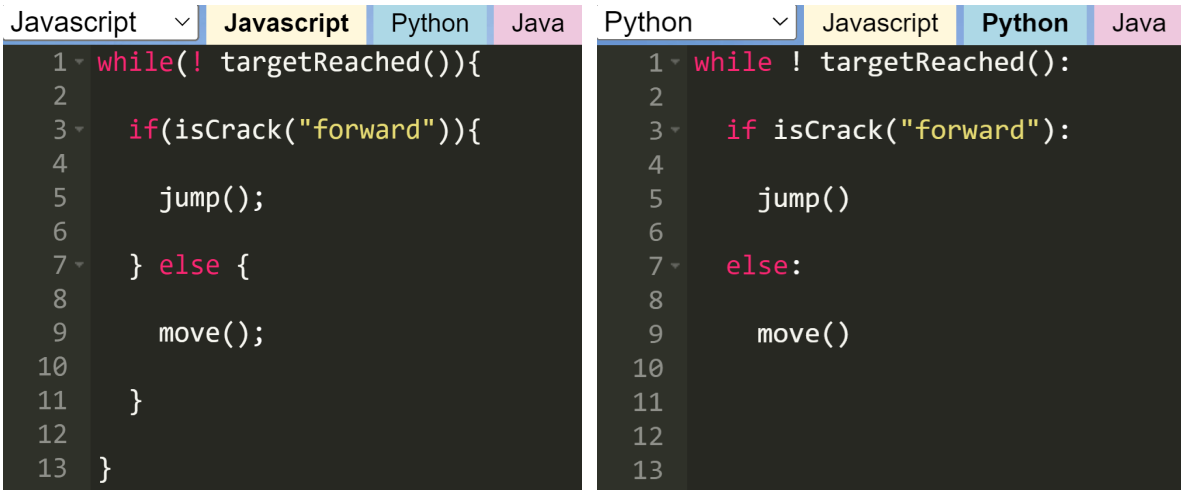


Figure 1.1 : codes sources dans deux langages différents

Uniformiser le choix du langage étudié restreint en revanche l'éventail de la nature des objets pouvant être manipulés par les programmes dont on analyse ou écrit le code. Un langage de programmation a souvent ses spécialités d'application, disposant d'un noyau ou de bibliothèques natives permettant de manipuler des types spécifiques d'objets. Le langage A peut par exemple constituer un bon outil afin de faire des mathématiques, de la statistique ou contrôler des robots alors que le langage B dispose des ressources nécessaires pour faire du graphisme et des animations en 3d, du traitement de données multimédia et de l'interfaçage web. Imposer un de ces langages a alors comme corollaire de renoncer aux champs d'application dans lesquels ce langage ne dispose pas des ressources nécessaires. Ce qui peut priver enseignants et élèves d'objets d'apprentissage potentiellement stimulants (afficher du texte et des nombres dans une console, dessiner des figures géométriques, concevoir un jeu vidéo ou une apps pour smartphone ne suscitent pas nécessairement le même degré de motivation chez tout le monde).

Comme si le dilemme du choix du langage de programmation idéal n'était pas suffisant, un collègue pose encore la question : visuel ou littéral ? En vous disant qu'il a fait de très bonnes expériences en commençant avec ses élèves l'apprentissage de la programmation à l'aide d'un outil tel que *Scratch* du MIT, avant de passer à un langage de programmation littéral. Les langages de programmation visuels, dont le code se présente le plus généralement sous la forme de blocs imbriqués représentant des instructions ou des éléments d'expression, constituent en effet une couche d'abstraction additionnelle aux langages littéraux de programmation de haut niveau (sous forme de texte), rendant la découverte de la programmation plus accessible, évitant aux débutants de se heurter aux problèmes de syntaxe, tout en se concentrant sur les fondamentaux. Il n'est pas rare que ce façon de voir provoque une levée de quelques boucliers jugeant ces langages trop

'enfantins' pour des gymnasiens ou des étudiants, même si ceux-ci n'ont encore jamais programmé de leur vie. Sans véritable argument de fond cependant.

Les enseignants et professeurs ayant déjà effectivement expérimenté et évalué l'usage d'un langage visuel tel que *Scratch* dans un cours d'introduction à la programmation ont des avis quelque peu contrastés et divergents sur le sujet. Ainsi, pour J. A. Martínez-Valdés et ses collègues (2017) de l'Université de Réy Juan Carlos, l'usage de *Scratch* dans un cours d'informatique de première année s'est révélée 'relativement insatisfaisante', pour reprendre le titre de leur article publié sur le sujet ('A (Relatively) Unsatisfactory Experience of Use of Scratch in CS1'). Après deux semaines de cours avec *Scratch*, ces enseignants ont ensuite basculé sur le langage *Java*. Ils en concluent que 'the results we obtained for both the Scratch language and the Dr. Scratch tool were less satisfactory than expected and, in some regards, disappointing.' En sens inverse, M. Armoni et al. (2015) arrivent à la conclusion opposée, en étudiant une même transition de *Scratch* vers *Java* ou *C#*, du secondaire 1 ou secondaire 2 :

> *We found that the programming knowledge and experience of students who had learned* Scratch *greatly facilitated learning the more advanced material in secondary school: less time was needed to learn new topics, there were fewer learning difficulties, and they achieved higher cognitive levels of understanding of most concepts (although at the end of the teaching process, there were no significant differences in achievements compared to students who had not studied* Scratch*). Furthermore, there was increased enrollment in CS classes, and students were observed to display higher levels of motivation and self-efficacy. This research justifies teaching CS in general and visual programming in particular in middle schools.*
>
> M. Armoni et al. (2015)
> *From Scratch to “Real” Programming*

Malan J. (2005) arrivait déjà à des conclusions similaire après avoir tenté l'usage de *Scratch* dans un cours d'introduction à l'informatique à l'Université de Harvard :

> *Although Scratch is intended to "enhance the development of technological fluency [among youths] at after-school centers in economically disadvantaged communities," we find remarkable potential in this programming environment for higher education as well. We propose Scratch as a first language for first-time programmers in introductory courses, for majors and non-majors alike. Scratch allows students to program with a mouse: programmatic constructs are represented as puzzle pieces that only fit together if "syntactically" appropriate. We argue that this environment allows students not only to master programmatic constructs before syntax but also to focus on problems of logic before syntax. We view Scratch as a gateway to languages like Java. [...] We find that, not only did Scratch excite students at a critical time (i.e.,, their first foray into computer science), it also familiarized the inexperienced among them with fundamentals of programming without the distraction of syntax. Moreover, when asked via surveys at term's end to reflect on how their initial experience with Scratch affected their subsequent experience with Java, most students (76%) felt that Scratch was a positive influence, particularly those without prior background. Those students (16%) who felt*

> *that Scratch was not an influence, positive or negative, all had prior programming experience.*
>
> Malan J. et al. (2005)
> *Scratch for budding computer scientists*

Aujourd'hui *Scratch* est toujours utilisé par J. Malan et ses collègues dans la première partie du cours d'introduction à l'informatique suivi par les étudiants de l'Université de Harvard (Malan J., 2022), ces professeurs étant visiblement convaincus de l'apport de l'apprentissage préalable d'un langage visuel avant un langage littéral. Quand on observe tout l'enthousiasme de Malan lorsqu'il donne son cours[1], il y a lieu de se demander si la motivation de l'enseignant à utiliser un moyen et un objet d'apprentissage spécifiques ainsi que la manière dont ils les utilisent ne constituent pas une variable ayant un impact significatif sur la performance des apprentissages. Et, a contrario, si l'imposition de moyens auxquels l'enseignant n'adhère pas ne péjore pas significativement cette même performance.

La recherche du langage de programmation idéal est un peu la quête du Saint-Graal de l'informaticien, relevant plus du fantasme que de la réalité. N'étant qu'un outil, on peut certes définir des critères d'éligibilité afin de ne retenir que les langages le plus à même d'atteindre l'objectif visé. Mais ceux-ci ne permettront la plupart du temps pas de restreindre l'ensemble du possible à un seul et unique langage, aboutissant au contraire à une liste de langages souvent proches.

Afin de concilier ces différentes visions du choix du langage de programmation à utiliser pour apprendre à programmer (langage visuel vs langage littéral, langage A vs langage B), peut-être est-il judicieux de prendre quelque peu du recul sur la notion de programme et de revenir à son origine, à savoir l'algorithme. Car au bout du compte, le code littéral ou visuel d'un programme n'est que l'expression sous forme spécifique d'un algorithme, sa représentation étant par définition polymorphe, tant ces formes possibles sont variées.

Un moyen d'apprentissage offrant la possibilité à l'élève ou l'étudiant de composer des algorithmes élémentaires sous différentes formes afin de manipuler à l'aide d'opérations simples un même objet peut offrir la perspective de dépasser ces divergences de points de vue avec le recul d'une vision globale et synthétique. Un tel instrument peut alors être utilisé par des élèves ayant atteint des stades d'apprentissage différents, en fonction de leur âge. De par le même objet manipulé par les programmes conçus, il facilite la transition d'un langage visuel à un langage littéral ainsi que la généralisation des fondamentaux des langages à travers la transposition d'un langage à un autre.

Peu nombreux sont de tels moyens d'apprentissage linguistiquement polymorphes d'apprentissage de la programmation à différents niveaux, ceux-ci n'étant pas simples à mettre en œuvre, se heurtant à des difficultés de transcodage. *Sauvez Julie*, l'application développée dans le cadre de ce travail de diplôme, est une modeste contribution à la réflexion et à l'expérimentation d'une telle approche polymorphe visant à concilier différentes doctrines tout en intégrant les parts de vérité que chacune comporte.

[1] https://www.youtube.com/watch?v=IDDmrzzB14M (02.08.2023)

2. Approche pédagogique et profils d'utilisateurs

Sauvez Julie est une application d'apprentissage à niveaux multiples de la programmation visuelle. Elle est accessible à des enfants découvrant la notion d'algorithme tout comme à des adolescents ou des adultes apprenant à passer de la représentation visuelle à la représentation littérale et formelle d'un algorithme.

2.1 Approche pédagogique

Tout instrument d'apprentissage repose, implicitement ou explicitement, sur une théorie de l'apprentissage, cette dernière mettant en lumière un processus de développement d'un savoir ou d'un savoir-faire.

a) objet d'apprentissage : de Piaget à Processing

L'une des théories de la cognition la plus reconnue est sans doute celle du psychologue suisse Jean Piaget (1896-1980). Ce dernier s'intéresse principalement au développement cognitif de l'enfant. La cognition peut être définie comme l'aptitude d'une personne à appréhender l'environnement dans lequel elle vit afin de pouvoir interagir avec celui-ci dans le but de satisfaire ses besoins et conséquemment d'y survivre et de s'y développer. La cognition est ainsi un processus de construction d'un modèle du monde dans lequel un individu vit. Pour Piaget, il s'agit d'un modèle dynamique en constante évolution s'adaptant aux changements de l'environnement.

Selon Piaget, le noyau cognitif se construit par couches tout au long de l'enfance, à travers différentes étapes correspondant au stade de développement de chacune des couches. Chaque couche se construit durant une période de développement de l'enfant. Piaget (1961) distingue quatre stades du développement cognitif de l'enfant (figure 2.1).

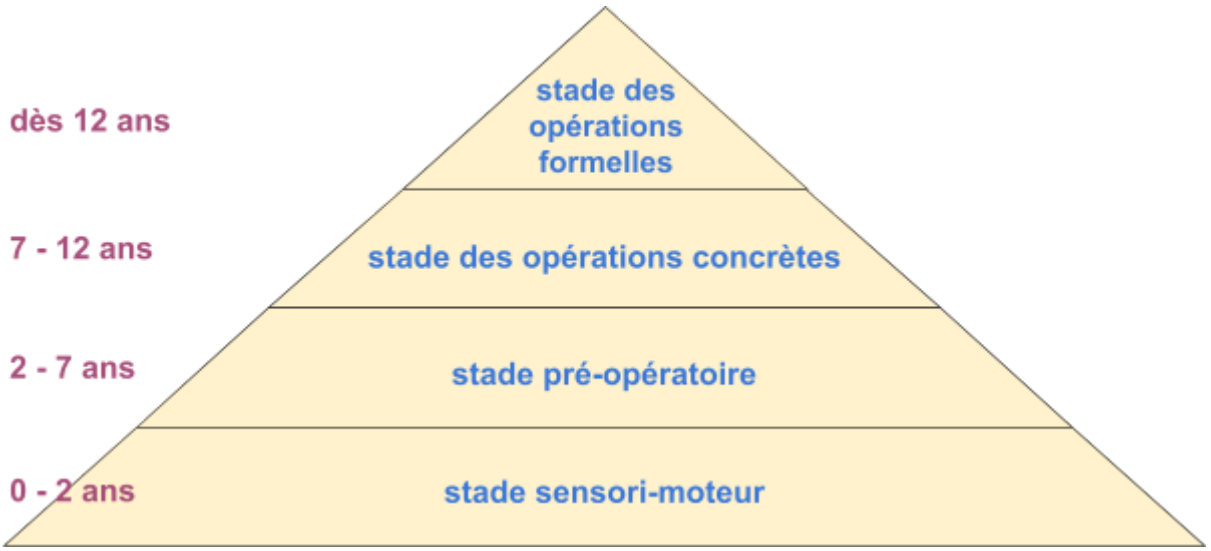


Figure 2.1 : stades du développement cognitif de l'enfant selon Piaget (Dubuc B., 2023)

Durant le *stade sensori-moteur*, entre ses premiers mois et sa deuxième année, l'enfant établit un contact en développant et exerçant ses capacités motrices, à l'image de celles de ses bras et mains. Il peut notamment saisir des objets. En le palpant, le sentant, le mettant à sa bouche ou écoutant les bruits qu'un objet manipulé fait, l'enfant le découvre expérimentalement en l'associant à ses attributs. Vers le milieu de ce stade, l'enfant développe la permanence des objets, à savoir sa capacité à réaliser qu'un objet continue d'exister lorsqu'il sort de son champ de perception visuelle.

Au cours de *stade pré-opératoire*, entre 2 et 7 ans, l'enfant commence à associer des objets à des symboles, qu'il s'agisse de mots ou d'images. Outre le développement d'une pensée symbolique, celle-ci commence à intégrer les notions de nombres (quantité), d'espace et de temps. L'enfant se met à faire une distinction entre les faits présents et futurs, même si sa pensée reste fortement ancrée dans le présent de son monde physique immédiat. Le symbolisme et la perception du futur, associés à la permanence des objets, le prépare au stade de développement suivant : le *stade des opérations concrètes*.

A ce stade, l'enfant devient à même de construire un modèle de son univers visible. Pour cela, il doit rester en contact avec celui-ci tout en continuant de l'expérimenter afin de développer son savoir. Son univers est perçu comme un ensemble d'objets ayant des liens entre eux. Ces liens entraînent des relations de causes à effets des actions entreprises sur ces objets. La compréhension et l'anticipation des relations causales des actions sur les objets permettent ainsi une relative maîtrise de leurs manipulations.

La généralisation et la catégorisation d'un type d'objet ou d'une relation de cause à effet dont l'enfant fait la découverte sont à l'origine de leur *assimilation* dans le grand puzzle que se construit l'enfant de son environnement direct. Ainsi, par exemple, l'enfant apprend que tout objet solide (catégorie) peut être saisi avec une main afin d'être déplacé (action). Chaque nouvel objet solide que l'enfant découvre sera ainsi assimilé à cette catégorie et le schème d'action associé, en permettant la manipulation selon un but précis, pourra lui être appliqué. Piaget entend par *schème d'action* une structure une "structure ou organisation des actions telles qu'elles se transfèrent ou se généralisent lors de la répétition de cette action en des circonstances semblables ou analogues" (Piaget, 1961). Un schème d'action s'applique donc à tous les objets d'un même type, pouvant lui être assimilée.

Que se passe-t-il lorsque l'enfant tente de manipuler un objet en suivant un schème d'action qui n'est pas approprié, ne lui permettant pas d'atteindre son but ? Si le schème d'actions pour déplacer un objet à une main convient à tout objet assimilable à un objet solide dont la forme comprend des angles (bords), il ne fonctionnera pas avec un objet sphérique. L'enfant ne parviendra pas à saisir un ballon avec une main pour le déplacer. Il ne pourra assimiler le ballon à la catégorie des objets solides anguleux. Il devra alors s'en accommoder. L'*accommodation* consiste en un remaniement du modèle opératoire que l'enfant a en tête, afin de lui permettre d'y intégrer une nouvelle catégorie d'objets (physique, mais par la suite également, au stade suivant, virtuel ou conceptuel) ainsi qu'un nouveau schème d'action ou l'adaptation d'un schème existant, lorsque la nouvelle catégorie est proche ou dérivée d'une catégorie existante. Afin d'y insérer une nouvelle pièce, l'enfant doit remanier le grand puzzle de son univers qu'il s'est construit jusqu'ici.

Au stade des opérations concrètes, l'enfant développe discrètement les prémices du stade suivant, à savoir sa capacité à faire de l'abstraction des éléments constituant son univers physique, avant de passer, durant le stade qui suit, aux éléments de son univers mental. Le dernier stade, celui des *opérations formelles*, débute aux alentours de 12 ans pour se poursuivre tout au long de la vie. A ce stade, l'adolescent devient à même de se construire des représentations formelles du monde. Les objets y sont catégorisés selon leur structure et leurs propriétés ainsi que représentés abstraitement par des symboles, tout comme les relations logiques existant entre ces objets. Cette représentation formaliste ouvre la porte de la représentation de phénomènes et d'objets non visibles directement dans le champ de perception de l'observateur. Avec l'imagination des évolutions possibles, les modèles formels s'animent et se transforment en simulations permettant de se projeter dans l'avenir par le biais de raisonnements hypothético-déductif, caractéristique de cet ultime stade du développement cognitif.

Pour Piaget, le développement cognitif de l'enfant se réalise dès les premiers mois et se poursuivra tout au long de sa vie dans ses interactions avec son univers non seulement par sa perception mais à travers la manipulation des objets le constituant : "Ce n'est donc pas seulement la perception mais bien l'action et les transformations que l'enfant peut opérer sur le monde qui lui permettent d'acquérir ses premières connaissances." (Dubuc, 2023). Piaget souligne ainsi l'importance fondamentale de l'apprentissage par la pratique.

Si Piaget a illustré sa théorie de l'apprentissage dans le domaine des mathématiques, elle n'en reste pas moins également des plus éclairantes sur l'apprentissage de la programmation. Quelles conclusions peut-on en tirer afin de rendre cet apprentissage plus efficace tout en évitant de s'égarer dans des pratiques moins performantes ? En donnant notamment des éléments de réponse à ces questions : à partir de quand et comment ?

A défaut de nous donner un repère quant à l'âge à partir duquel un enfant est à même d'appréhender la notion d'algorithme et d'apprendre à programmer, la théorie de l'apprentissage de Piaget nous renseigne sur la période à éviter. Si celle-ci inclut sans aucun doute les deux premières années de vie, d'aucuns s'interrogeront sur le stade suivant, celui de la pensée pré-opératoire. N'y a-t-il pas déjà moyens d'y faire découvrir quelques notions, même rudimentaires, à l'instant de celles d'opérations et de séquences d'instructions ? Et ceci à partir de 4 ou 5 ans, au niveau des premières années de scolarité, à l'école enfantine.

Sans doute cela est-il envisageable. Mais il y a lieu de tenir également compte des autres apprentissages fondamentaux qu'un enfant est appelé à réaliser durant ces premières années de scolarité. Et de les hiérarchiser compte-tenu de ressources d'apprentissage, à commencer par le temps et l'énergie à disposition, forcément limités. Selon Piaget, la principale compétence développée durant le deuxième stade est celle du langage naturel de l'humain, et non celui formel des ordinateurs, nécessitant a priori des aptitudes voyant le jour dans les stades ultérieurs.

Introduire des activités d'apprentissage de l'algorithmique ou de la programmation tend à être ainsi prématurées au niveau de la crèche ou de l'école enfantine si celles-ci font ombrage au développement d'autres aptitudes plus fondamentales pour cette tranche d'âge. Et cela d'autant plus si ces apprentissages devaient se faire sur écran. Une abondante littérature met en outre en évidence les effets contre-productifs d'un usage excessif de

l'apprentissage avec les technologies numériques, par le biais d'écrans (Toyama K. (2015), Bihouix P. et Mauvilly K. (2016), Spitzer M. (2019), Patino B. (2019), Desmurget (2020), Marry Y. et Souillot F. 2022). Une vaste étude menée par l'OCDE en 2015, dans les écoles de ses pays-membres, en parallèle à l'étude PISA, confirme ce constat. Schleicher A. et al. (2015) résument ainsi les conclusions de leurs études :

> *[...] même lorsque les nouvelles technologies sont utilisées en classe, leur incidence sur la performance des élèves est mitigée, dans le meilleur des cas. [...] Les élèves utilisant très souvent les ordinateurs à l'école obtiennent des résultats bien inférieurs dans la plupart des domaines d'apprentissage. [...] Selon les résultats de l'enquête PISA, les pays qui ont consenti d'importants investissements dans les TIC dans le domaine de l'éducation n'ont enregistré aucune amélioration notable des résultats de leurs élèves en compréhension de l'écrit, en mathématiques et en sciences. [...] En un mot, le fait de garantir l'acquisition par chaque enfant d'un niveau de compétences de base en compréhension de l'écrit et en mathématiques semble bien plus utile pour améliorer l'égalité des chances dans notre monde numérique que l'élargissement ou la subvention de l'accès aux appareils et services de haute technologie.*
>
> Schleicher A. et al. (2015), p. 1
> *Connectés pour apprendre ? les élèves et les nouvelles technologies* (Paris : OCDE)

Les technologies et les pratiques numériques évoluant rapidement, il est légitime de se demander si cet état des lieux de l'apprentissage à l'aide du numérique est toujours pertinent. La littérature citée précédemment tend à étayer une réponse par l'affirmative. Tout comme les faits, avec la décision au printemps 2023 de la Suède de revenir sur sa stratégie de numérisation de l'enseignement mise en oeuvre depuis plus de 10 ans. Constatant une diminution significative des aptitudes fondamentale de ses écoliers suite au suivi de ce programme, le gouvernement suédois a décidé de faire marche arrière en investissant plusieurs dizaines de millions de francs afin que tous les élèves suédois disposent à nouveau d'un manuel papier dans chacune des disciplines étudiées (Hivert A.-F. , 2023).

Les conclusions des chercheurs et de l'OCDE ne s'appliquent pas qu'à l'école maternelle ou enfantine mais concernent l'ensemble de la scolarité obligatoire. Ainsi, si la découverte de l'algorithmique et de la programmation ne sont manifestement pas appropriés au stade pré-opératoire, de 2 à 7 ans, il s'agit également d'être prudent aux stades ultérieurs, de l'école primaire au cycle d'orientation : on songera à un usage ciblé et limité des écrans, en y intégrant dans les apprentissages des activités analogiques sans écran.

Le stade des opérations concrètes (7 à 12 ans), à savoir celui correspondant à la troisième année de l'école primaire jusqu'à la première année du cycle d'orientation) tend à enfin ouvrir les portes d'une première découverte de la notion d'algorithme. Après avoir acquis les premières bases du langage naturel, capable de travailler avec des symboles et commençant à développer les prémisses d'une capacité d'abstraction en mathématique (nombres, opérations arithmétiques, notions de géométrie, etc). L'enfant commence à être à même de conceptualiser son monde à travers des opérations mentales. Celles-ci doivent cependant rester en contact direct avec une réalité tangible. L'enfant développe de la sorte

ses aptitudes cognitives en interagissant avec son univers physique, en manipulant les objets le constituant. Un enfant apprendra par exemple, dans un premier temps, à faire des additions en s'aidant de ses doigts afin de compter.

Si l'on se fie à la théorie de Piaget, la notion d'algorithme est intrinsèquement ancrée dans le développement cognitif. En effet, ce que Piaget désigne comme un schème d'action peut s'apparenter en programmation à une fonction à même de manipuler des objets d'une même classe. Dans l'optique piagétienne, une telle fonction est capable de s'adapter à l'évolution de son environnement par accommodation à la nature des objets sur lesquels elle permet d'effectuer des opérations (ce que serait capable de faire une fonction faisant partie d'une intelligence artificielle).

Il n'en reste pas moins qu'à la base, un schème d'action n'est rien d'autre qu'une structure d'opérations permettant de manipuler un objet afin d'atteindre un objectif déterminé. L'essence même d'un schème d'action est donc un algorithme. Sans le savoir, aussi bien un enfant qu'un adulte ne cesse de développer et de mettre en oeuvre de nouveaux schèmes d'action en interagissant avec leur univers. Tel Monsieur Jourdain faisant de la prose, un enfant pratique de la sorte l'algorithmique sans le savoir. C'est là un point de départ et un atout indéniable dans l'apprentissage des notions d'algorithmique et de programmation.

Le stade des opérations concrètes est donc un terreau fertile propice à la découverte de l'algorithmique. A condition cependant que les notions élémentaires d'algorithmique (opération, séquences, répétitions, branchements) s'acquièrent en étudiant ou en concevant des algorithmes constitués d'opérations concrètes de manipulation d'objets du monde physique ou visuel de l'enfant. Si une première approche de l'algorithmique peut se faire à l'aide d'activités débranchées (description ou suivi d'un itinéraire, réalisation d'une recette de cuisine, tri de camarades alignés du plus petit au plus grand, etc), la découverte de l'ordinateur et l'apprentissage de la programmation seront sans doute plus motivants sur un écran. En élaborant de petits programmes dont l'exécution se manifeste par des résultats physiquement observables et motivants.

C'est que s'est dit Seymour Papert (1928-2016), professeur de mathématiques et d'informatique, disciple de Jean Piaget avec qui il a collaboré et dont il a adopté sa vision pédagogique constructiviste de l'apprentissage, fondée sur sa théorie de l'apprentissage. Aussi bien pour Piaget que pour Papert, un enfant construit expérimentalement son savoir dans ses interactions avec son environnement. Papert met en exergue le postulat piagétien selon lequel c'est non seulement en expérimentant que l'on apprend, mais plus précisément en créant. "C'est en créant qu'on apprend" est la thèse qu'il soutient notamment dans son célèbre ouvrage de référence "Le jaillissement de l'esprit : ordinateurs et apprentissage" (Papert, 1981) dans lequel il cherche à démontrer la plus-value d'une apprentissage actif à l'aide d'un ordinateur.

Au niveau de la programmation, ce lien entre apprentissage et création peut prendre la forme d'un robot dont le déplacement est contrôlé par l'exécution d'un programme écrit dans un langage de programmation proche du langage naturel, facilement assimilable par une personne n'ayant jamais programmé car utilisant des mots et une syntaxe proche du langage humain. Telles sont certaines des idées ayant germé dans la tête de Papert et de ses collègues dès la fin des années '60 au MIT Computer Science and Artificial Intelligence

Laboratory à Boston. Celles-ci donnèrent naissance au célèbre langage de programmation *Logo* ainsi qu'à la tortue graphique, un robot physique ou virtuel dessinant un trait sur une feuille ou un écran.

La version physique du robot se présente sous la forme d'une tortue sur roues dont les mouvements sont ordonnés par l'exécution d'un programme. A sa base, la tortue dispose d'un stylo-feutre touchant le sol sur lequel le robot se déplace. Le déplacement du robot, se comportant comme une main tenant un crayon, dessine les traits constitutif d'un dessin (figure 2.2)

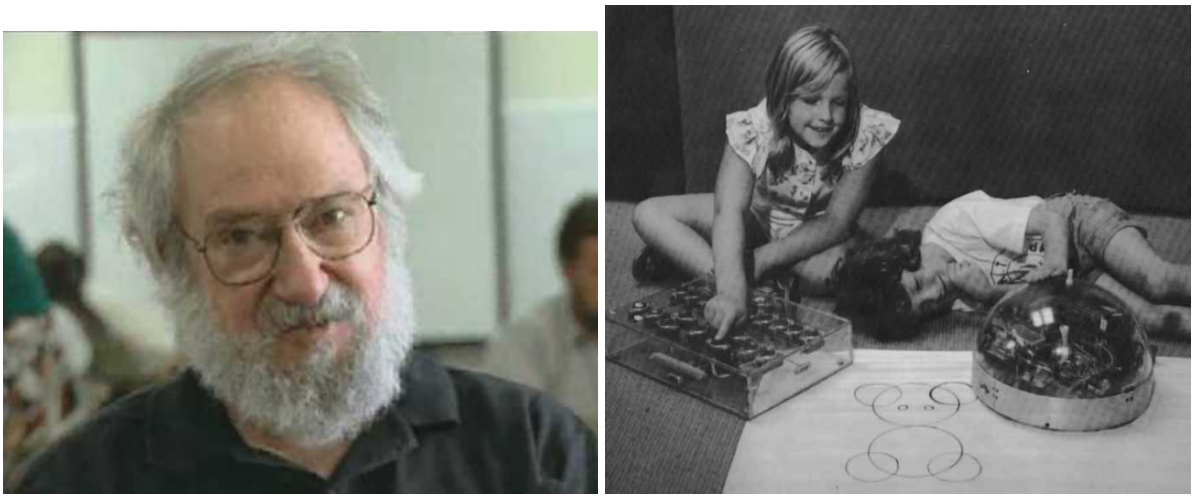


Figure 2.2 : Seymour Papert et la tortue *Logo*

Le langage *Logo* est souvent assimilé à un langage permettant juste de piloter cette tortue alors qu'il s'agit d'un langage de programmation complet capable de gérer des structures de données telles des listes ainsi que de les enregistrer dans des fichiers.

Depuis le milieu des années '60 jusqu'à la fin des années '70, le langage *Logo* a connu une longue phase d'incubation durant laquelle il s'est développé et a été expérimenté avec des élèves, avant de connaître son apogée et de se généraliser au cours dans la première moitié des années '80 avec l'émergence des ordinateurs personnels dans les foyers (Wikipédia, 2023), avant de péricliter pour renaître sous de nouvelles formes dès les années '90.

Si la tortue *Logo* existe bien sous la forme de robots physiques, c'est sa version virtuelle, sur un écran, qui a toujours été privilégiée pour des raisons logistiques et financières évidentes (expliquant également l'usage limité des robots dans les écoles) (figure 2.3). Cette tortue suit au ralenti les instructions qu'on lui donne, ce qui permet de voir son déplacement sur l'écran lors de la production du dessin. Cette caractéristique du langage *Logo* le distingue des autres langages de programmation dont les instructions sont effectuées instantanément à l'exécution du programme, ce qui ne permet pas au programmeur débutant de voir les opérations se réaliser les unes après les autres, à moins de faire du pas à pas dans un débogueur. Avec des langages de programmation conventionnels, le programmeur débutant obtient habituellement instantanément le produit final issu de l'exécution de son programme, alors qu'avec *Logo* celui-ci se construit progressivement sous ses yeux. Ce qui va l'aider à

identifier visuellement et corriger ses erreurs, un élément particulièrement important de l'apprentissage piagétien.

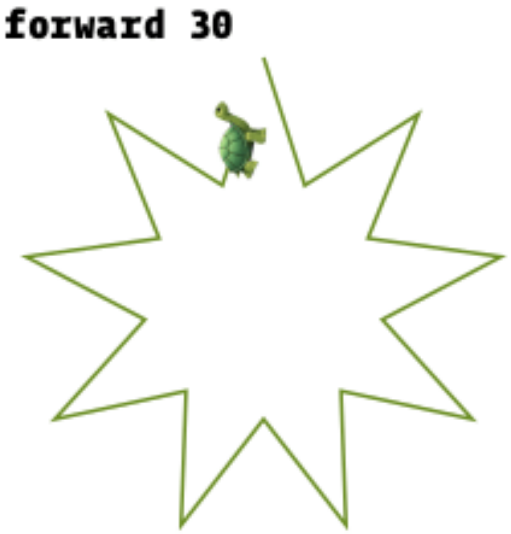


Figure 2.3 : tortue *Logo* sur écran

La tortue *Logo* s'est métamorphosée par la suite en objets visuels (images) multiples se déplaçant sur écran, dont le pilotage ouvre la possibilité de création d'animations et de jeux vidéo. Au début des années '80 déjà, des ordinateurs grand public tels que le Commodore VC-20 permettaient d'écrire en *Basic* ou en *assembleur* des programmes animant des *sprites* (lutins) à l'écran, acteurs virtuels d'une multitude de jeux vidéo d'arcade. Plutôt que de dessiner des figures géométriques, l'apprenti-programmeur conçoit alors des dessins animés ou des jeux vidéo, ce qui accroît d'autant sa motivation.

On oublie fréquemment que Papert et ses collègues ont développé la tortue *Logo* dans l'intention d'offrir aux élèves et à leur enseignants une nouvelle manière, fondée sur le constructivisme de Piaget, d'apprendre par la pratique des notions de géométrie. L'objectif premier de Papert n'était pas l'apprentissage de la programmation. Or, de nos jours, la tortue *Logo* est le plus souvent assimilée à un outil d'apprentissage pour débutant de l'algorithmique et de la programmation.

Utilisée à cette fin, cette approche peut rapidement se heurter à ses limites et être abandonnée après quelques heures de cours. Tout d'abord, celle-ci peut générer de la lassitude face à un type de problème répétitif (concevoir des figures géométriques). Mais ensuite, et surtout, elle cumule l'apprentissage simultané de concepts de deux domaines différents, à savoir les mathématiques et l'informatique. Si ceux-ci sont certes complémentaires et qu'une approche combinée peut a priori amener un surcroît de motivation et une plus grande efficacité d'apprentissage à travers leur synergie, l'expérience tend à montrer que dans le cadre de la tortue *Logo*, les concepts de programmation et de géométrie ne s'intègrent pas aussi aisément chez des élèves voyant au contraire le poids de

leur difficulté d'assimilation s'accumuler, ce qui peut finir par les décourager. Si un élève doit, en plus de notions d'algorithmique et de programmation, apprendre le théorème de Thalès ou de Pythagore afin d'écrire le code du programme lui permettant d'obtenir la figure géométrique attendue, il peut rapidement, comme débutant en programmation, saturer face aux difficultés multidisciplinaires rencontrées.

Du côté du MIT, d'autres personnes ont creusé l'idée d'apprendre la programmation en produisant des contenus visuels sur écran. Les technologies numériques ayant évolué et leur capacité s'étant fortement accrue (processeurs, cartes graphiques et cartes-son), d'aucuns se sont dits que l'on pourrait générer des objets plus motivants que de simples figures géométriques. Tout en partant de l'idée que les étudiants en arts visuels, en tant que futurs designers, architectes ou artistes, se devaient également de savoir programmer, ne pouvant se cantonner à l'utilisation d'outils courants de production graphique, souvent fort coûteux ou limitant leur créativité. Cette volonté de formation des artistes à l'informatique anticipait déjà les futurs besoins d'une industrie naissante du graphisme numérique. Mais pour cela, encore fallait-il (et faut-il toujours) apprendre à programmer à des étudiants ne se destinant pas à des études en informatique (Surguy M., 2018) :

> *Tools like Photoshop were built by large corporations and did not provide ways for regular users to quickly program some new features, while programming environments like Logo were too limited in their capabilities. In other words, if you were to learn programming graphics in the 1990s, you would need to know quite a bit of computer science in order to draw and animate a few hundred circles on the screen, or you would need to pay thousands of dollars for software that could do this for you in a limited fashion. In order to remove this gap and to create a more user-friendly way of visual programming, new ways of programming were explored at research institutions like MIT.*
>
> Surguy M. (2018), The History of Processing : Introduction to generative arts and Processing

De la sorte, différents projets ont vu le jour au Media Lab du MIT, à commencer par l'application *Design By Numbers* en 1999, environnement de programmation graphique développé par John Madea (2001), embryon sur lequel va être conçu le projet *Processing*. Les fondateurs de ce dernier, Benjamin Fry et Casey Reas, étudiants de Madea, se sont vus confrontés au problème de rendre un langage de programmation utilisé par l'industrie accessible à des débutants. Optant dès le départ pour une solution multi-plateformes, téléchargeable, portable et pouvant s'exécuter à la fois en version desktop ainsi que dans un navigateur web, ces deux artistes et informaticiens ont opté pour le langage Java, réputé pour être difficilement abordable par des débutants.

Outre un degré d'abstraction élevé nécessitant la maîtrise de passablement de notions de programmation afin de dessiner un simple cercle dans une fenêtre, *Java* contraint le programmeur à maîtriser le paradigme orienté objet. Cela n'a pas pour autant découragé Fry et Reas, qui se sont mis à la tâche en développant dès 2001 un IDE *Java* élémentaire dissimulant l'aspect orienté objet du langage, accompagné d'une bibliothèque offrant des

instructions avec lesquelles il devient possible de dessiner sur une fenêtre, avec quelques instructions simples, des figures géométriques complexes en 2 et 3 dimensions.

Un projet *Processing* (dénommé 'sketch' ou 'croquis') est constitué d'un code source comprenant par défaut une fonction *setup*() dans laquelle sont placées les instructions exécutées au démarrage du programme. Celles-ci créent tout d'abord un canevas sur lequel le programme dessinera ce qui lui est demandé, selon les instructions lui étant données (figure 2.4). Le code du projet peut être complété par une fonction *draw*() exécutée par défaut tous les 60^{e} de seconde, permettant de redessiner le canevas à cet intervalle et de générer ainsi des dessins animés et autres animations. *Processing* dispose de fonctions pour dessiner et animer des figures en 2 ou 3 dimensions, de générer et de traiter des sons ainsi que de lire, traiter et afficher des flux multimédias issus de fichiers audios ou vidéos ou de périphérique tels de webcams ou des micros (figure 2.5).

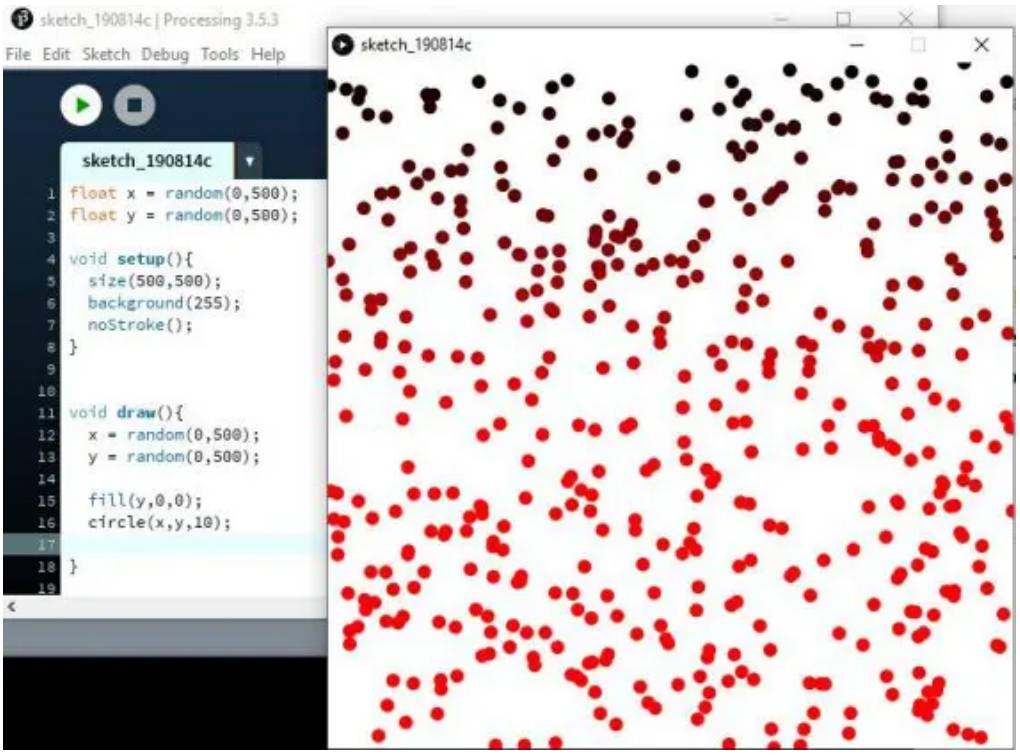


Figure 2.4 : IDE desktop Java, croquis et canevas *Processing*

Dès le départ, Casey Reas a misé sur un projet opensource documenté dont le développement a été confié à une communauté de volontaires. Les utilisateurs de *Processing* ne cessèrent d'augmenter. *Processing* s'est fortifié de par une intense concurrence avec l'environnement de programmation multimédia *Flash* de Macromedia, aujourd'hui disparu.

Figure 2.5 : *Processing*, animation d'un cube avec texture bitmap

Processing s'est depuis élargi en intégrant d'autres langages de programmation, avec l'ajout de bibliothèques constituées des mêmes instructions que celle de la version *Java* d'origine. Il devient ainsi possible d'écrire un même code source dans différents langages de programmation : *Java*, *Python* et *Javascript*. Un IDE en ligne a également été développé et mis à disposition de tout utilisateur gratuitement (https://editor.p5js.org/). Dans sa version *Javascript*, un projet est constitué d'une page web au sein de laquelle le canevas est inséré comme éléments *HTML*. La page *HTML* peut être utilisée comme interface utilisateur graphique, la bibliothèque *p5js* mettant à disposition des fonctions servant à manipuler les éléments *HTML* de l'arborescence *DOM* et les styles *CSS* leur étant appliqués. C'est là un atout important dont ne disposent pas les versions *Java* ou *Python*, la définition d'une interface graphique étant sensiblement plus complexe avec ces langages orientés desktop.

Les environnements technologiques dans lesquels *Processing* peut être utilisé se sont également étendus : en plus *Windows*, *MacOS* et *Linux* pour sa version desktop, les navigateurs web pour la version web, des projets *Processing* peuvent également être développés sous *Android* (smartphones, tablettes, montres) ainsi que pour piloter des contrôleurs *Arduino* ou *Raspberry Pi*.

Fort d'une communauté n'ayant cessé de s'élargir durant plus de 20 ans, le projet *Processing* s'est mué depuis 2012 en une fondation en charge de la promotion de son développement et de sa distribution. Daniel Shiffman, professeur d'informatique à l'Université de New-York, a rejoint le conseil de fondation et les leaders de l'équipe de développement de *Processing*, et plus particulièrement de sa version *Javascript* (Shiffman, 2012). Il alimente

depuis des années un site web et une chaîne personnelle *Youtube* au sein de laquelle il publie de nombreuses séquences d'apprentissage de la programmation avec *Processing* (https://thecodingtrain.com/).

De nos jours, *Processing* est également utilisé dans le monde professionnel par les artistes, les designers, les producteurs multimédias et les musées. Les premières oeuvres réalisées à l'aide de *Processing* ont impressionné les experts du domaine, au point que certaines ont été exposées dans des musées en Europe, au Japon et en Amérique, à l'image de celles Jared Tarbell (figure 2.6) ci-dessous :

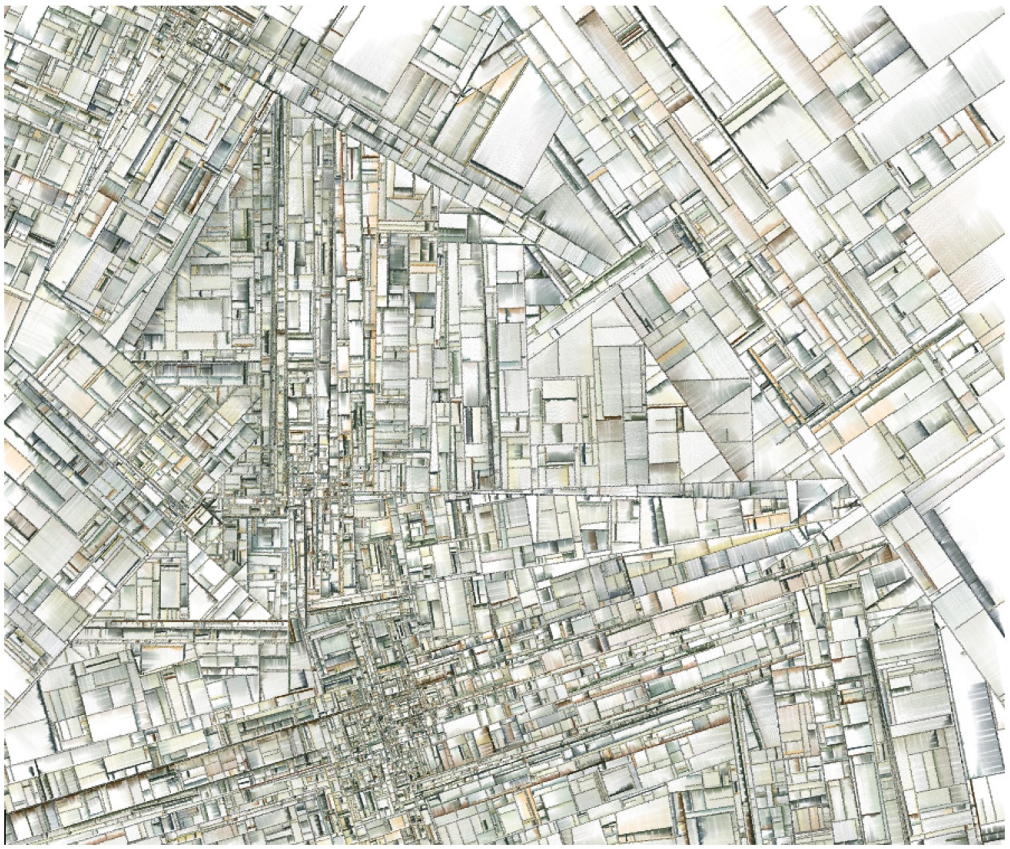


Figure 2.6 : *Substrate*, tableau de Jared Tarbell créé avec *Processing* (2003)

Aujourd'hui, les IDE et les bibliothèques constituant le framework de *Processing* sont utilisés dans des nombreuses écoles, y compris des universités et des écoles d'ingénieurs pour l'enseignement de la programmation selon une optique piagétienne. *Processing* peut de la sorte être vu comme une suite naturelle à *Logo* pour l'apprentissage de la programmation à partir du dernier stade d'apprentissage de la théorie de Piaget, celui des opérations formelles, à partir de 12 ans. En effet, la conception et la manipulation d'objets plus complexes, tels des images vectorielles 3d, à travers l'écriture de code dans un langage conventionnel de haut niveau, nécessite un degré d'abstraction le plus souvent insuffisamment développé au stade des opérations concrètes.

Sauvez Julie opte également pour une approche constructiviste : le joueur est appelé à composer un itinéraire permettant à un guide de montagne de suivre un chemin sûr afin de rejoindre une touriste qu'il doit secourir, en évitant de tomber dans une crevasse ou d'être victime d'une chute de rocher (figure 2.7). Comme dans le cas de la programmation *Logo*, le joueur voit son programme s'exécuter au ralenti, à travers le déplacement séquentiel du guide sur le glacier, ce qui lui permet de plus aisément détecter et corriger les erreurs de celui-ci.

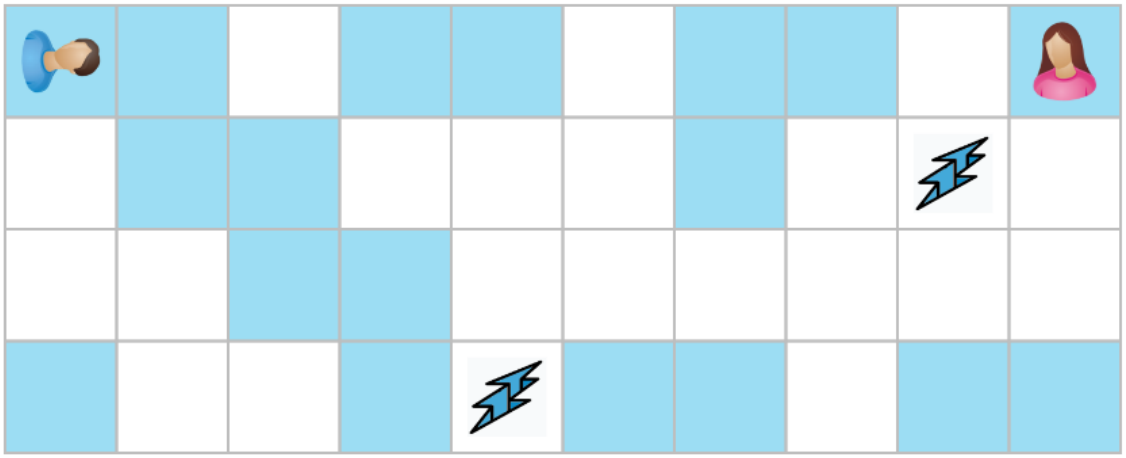


Figure 2.7 : scène de jeu de *Sauvez Julie*

Si l'apprentissage par la pratique se fait à travers la manipulation d'objets, le stade d'apprentissage est déterminant quant au degré de complexité des objets pouvant être manipulés par la personne apprenante. On ne manipule pas un hochet comme on joue de la guitare, on ne conduit pas un tricycle comme on pilote un avion de ligne. Il en va de même avec les objets virtuels. Déplacer une tortue *Logo* afin de dessiner un carré sur un écran nécessite des aptitudes élémentaires de moindre niveau que de concevoir et d'implémenter une base de données ou de définir et animer une image vectorielle en 3 dimensions.

Cependant, la nature de l'objet manipulé ne permet pas de déterminer à elle seule le stade d'apprentissage approprié. Encore faut-il encore tenir compte du degré de complexité de l'instrument utilisé afin de manipuler l'objet.

b) moyens d'apprentissage : du visuel au littéral

On peut apprendre à construire des armoires à l'aide d'un marteau et d'une scie ou avec un robot industriel qu'il s'agit de configurer et de programmer. Les instruments utilisés constituent également des moyens d'apprentissage servant à manipuler ou créer l'objet d'apprentissage. La manipulation d'un instrument de production est un objet d'apprentissage à proprement parler. Ainsi, la nature de l'instrument va déterminer le stade devant être atteint afin de permettre l'apprentissage de son maniement.

Par exemple, un enfant peut utiliser un crayon, une règle et un compas afin d'apprendre à dessiner et mesurer des figures géométriques élémentaires. La production et la manipulation de ces objets d'apprentissage est l'occasion pour l'enfant d'en découvrir leurs définitions ainsi que leurs propriétés. Mais le maniement du crayon, de la règle et du compas est également un objet d'apprentissage, appartenant au stade des opérations concrètes de Piaget, tout comme l'apprentissage de notions élémentaires de géométrie. Qu'en est-il lorsque l'on apprend ces mêmes éléments de géométrie en utilisant une tortue dont on programme les mouvements ?

Cela dépend vraisemblablement de la nature du langage de programmation utilisé et de sa proximité avec le langage naturel, tant du point de vue, de sa forme, de son vocabulaire que de sa syntaxe et de sa grammaire. Si cette proximité est grande, comme dans le cas du langage *Logo*, le stade d'apprentissage des opérations concrètes suffira sans doute. En revanche, comme déjà discuté au point précédent, si l'on a affaire à un langage de programmation conventionnel de haut niveau, on peut légitimement s'attendre à ce que l'enfant l'apprenant ait atteint le stade d'apprentissage des opérations formelles.

Au bout du compte, dans la détermination du stade d'apprentissage minimal prérequis pour l'usage d'un moyen d'apprentissage, il y a lieu de choisir le stade le plus petit en adéquation avec le moyen d'apprentissage lui-même et son objet. Dans le cas d'une tortue graphique pilotée à l'aide d'un programme écrit en *Logo*, il s'agira du stade des opérations concrètes, alors que pour un programme écrit en *Java* par exemple, on peut s'attendre à ce que le stade minimal soit celui des opérations formelles.

Un moyen d'apprentissage de l'algorithmique peut prendre différentes formes, l'algorithme étant lui-même un objet polymorphe pouvant être représenté sous une forme visuelle (ordinogramme, structogrammes, etc) ou littérale (description dans un langage naturel, pseudo-code, code source écrit dans un langage de programmation). Si l'algorithme est représenté comme un programme qu'un ordinateur est à même d'interpréter ou compiler, ce dernier pourra alors l'exécuter. Lorsque son exécution se traduit par une manipulation d'un objet physique (robot, tortue graphique) ou virtuel (crayon, lutin, pion sur un échiquier, etc) produisant une animation, l'enfant peut alors visualiser la mise en oeuvre de l'algorithme, ce qui lui donne un feedback offrant la possibilité de détecter et corriger ses erreurs.

Afin de bénéficier de cette rétroaction nécessaire à une approche constructiviste fondée sur la théorie de l'apprentissage de Piaget, un moyen d'apprentissage de l'algorithmique doit permettre l'élaboration d'un algorithme sous une forme exécutable, ce qui implique que ce dernier soit formulé dans un langage de programmation. Il est possible de classifier les langages de programmation en deux catégories : les langages de programmation littéraux et les langages de programmation visuels. Les langages littéraux permettent de coder des programmes sous forme textuelle alors qu'avec les langages visuels le programme est codé avec des figures géométriques, usuellement appelés 'blocs'.

Scratch, l'application d'apprentissage de la programmation visuelle pour débutant développée au sein du Media Lab du MIT depuis 2003 (Wikipédia, 2023) – actuellement la plus utilisée au monde par les enfants et leurs enseignants – fait usage de tels blocs afin de créer des programmes produisant des animation multimédia et des jeux vidéos constitués de sprites (lutins), images d'acteurs se déplaçant sur une scène virtuelle (figure 2.8)

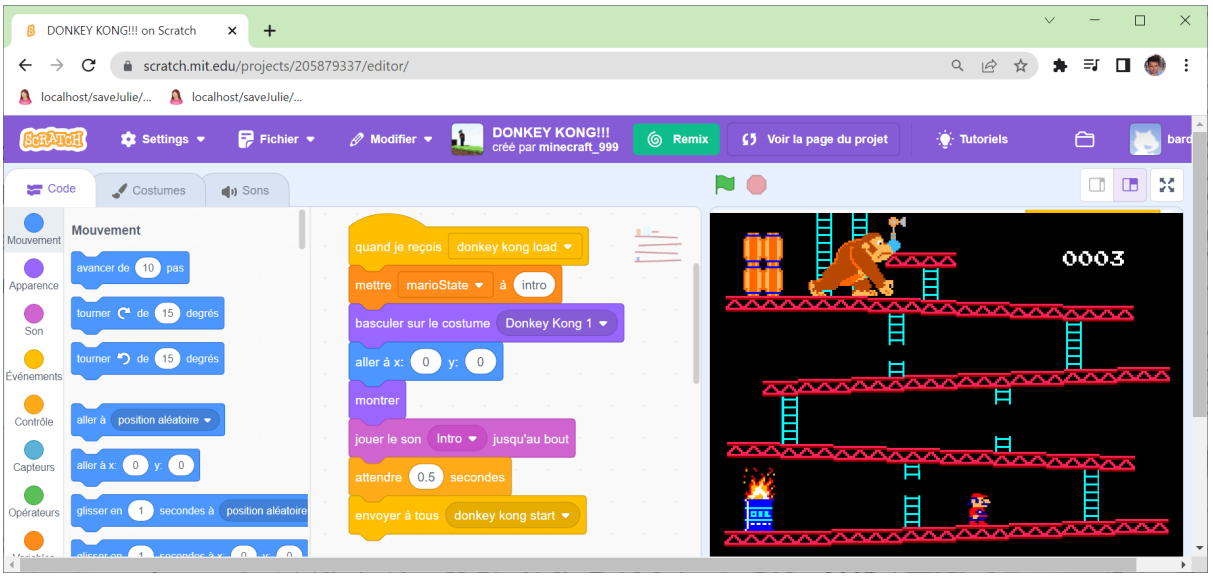


Figure 2.8 : application de programmation visuelle Scratch

Les langages de programmation visuels constituent une couche d'abstraction reposant sur une couche inférieure à laquelle la couche supérieure fait appel. La couche des langages visuels repose sur la couche des langages de programmation littéraux de haut niveau, laquelle s'appuie sur la couche des langages assembleurs se fondant elles-mêmes sur la couche des langages machines (schéma figure 2.9).

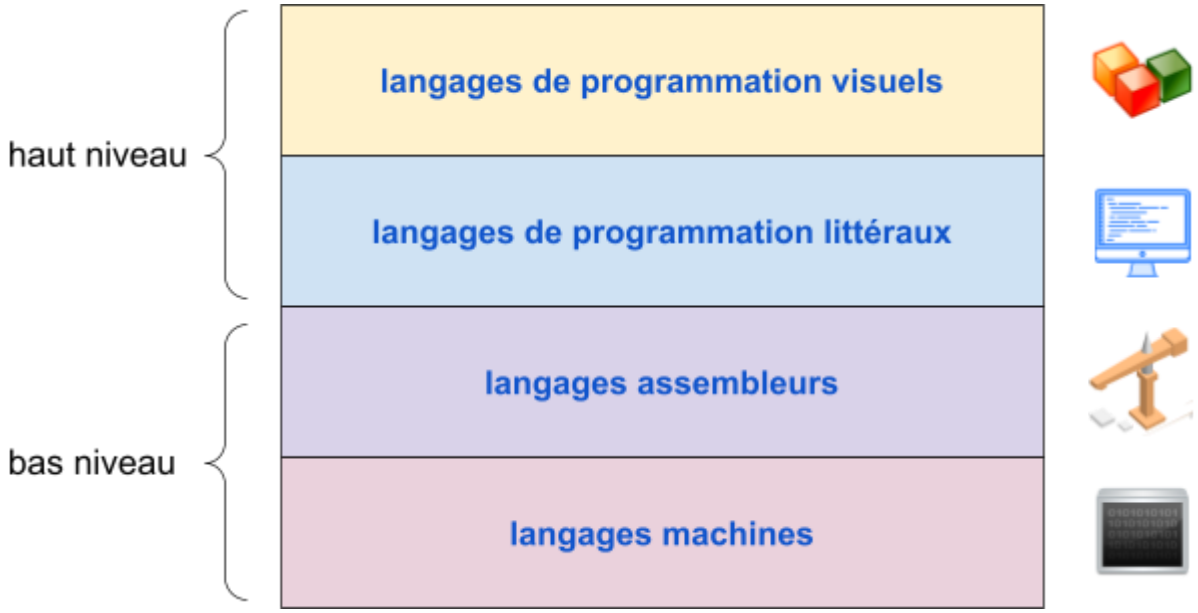


Figure 2.9 : modèle en couches des langages de programmation

Afin d'être exécuté par un ordinateur, un programme écrit dans un langage visuel devra donc d'abord être converti dans un langage littéral de haut niveau à l'aide d'un générateur de code (voir point 3.2.b et figure 2.10 ci-dessous). Celui-ci est à son tour traduit en un assemblage d'instructions machines symboliques elles-mêmes représentées sous forme binaire en langage machine exécutable par le microprocesseur d'un ordinateur.

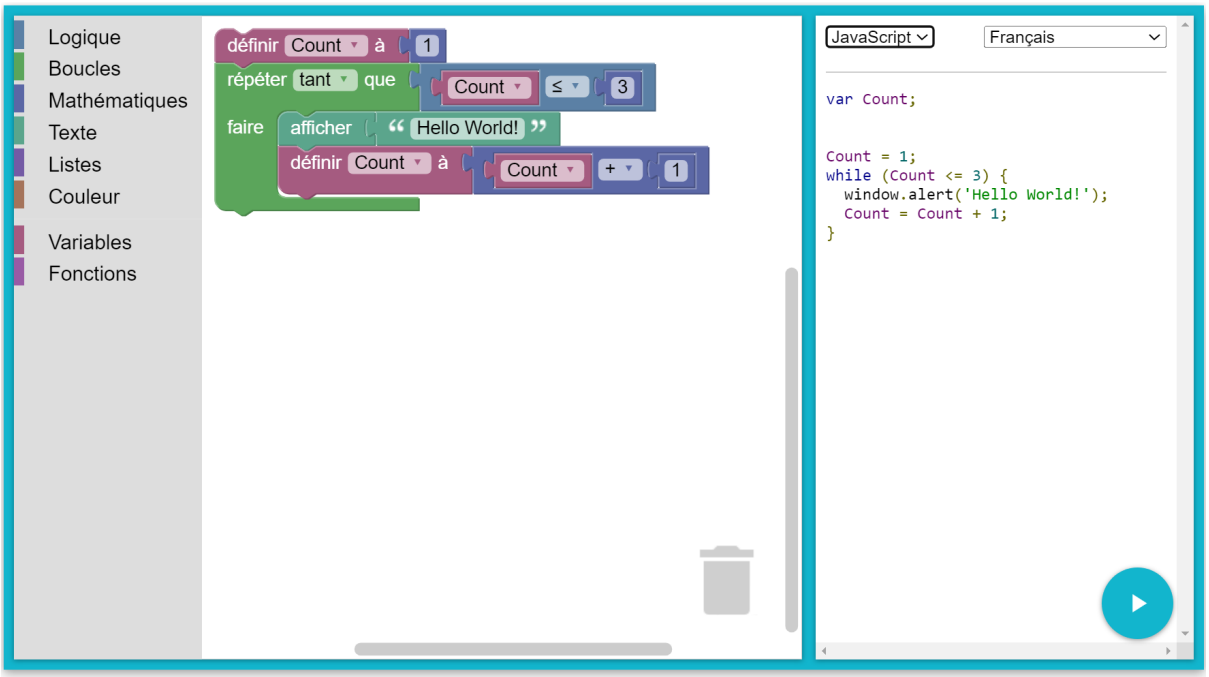


Figure 2.10 : génération de code visuel à partir du code littéral

L'utilisation d'un langage visuel comme moyen d'apprentissage de l'algorithmique et de la programmation offrant la possibilité de manipuler des personnages et des objets sur un écran met cet apprentissage à portée d'un enfant de 7 ans ayant atteint le stade piagétien des opérations concrètes. A tout le moins pour des manipulations élémentaires car, comme nous l'avons déjà vu, plus le degré de complexité de l'objet d'apprentissage augmente, plus l'enfant doit avoir un niveau d'apprentissage élevé.

Le degré de complexité du moyen d'apprentissage n'est pas déterminant à lui seul. Ainsi, faire se déplacer sur scène en ligne droite un dragon avec un instrument visuel tel que *Scratch* est à la portée d'un enfant au stade des opérations concrètes alors que, toujours avec *Scratch*, gérer des activités concurrentes comme par exemple des voitures allant en sens inverse et devant se partager la voie unique d'un pont pour traverser une rivière (figure 2.11) est un objet d'apprentissage plus complexe, nécessitant et degré d'abstraction et l'apprentissage d'aptitudes du stage des opérations formelles.

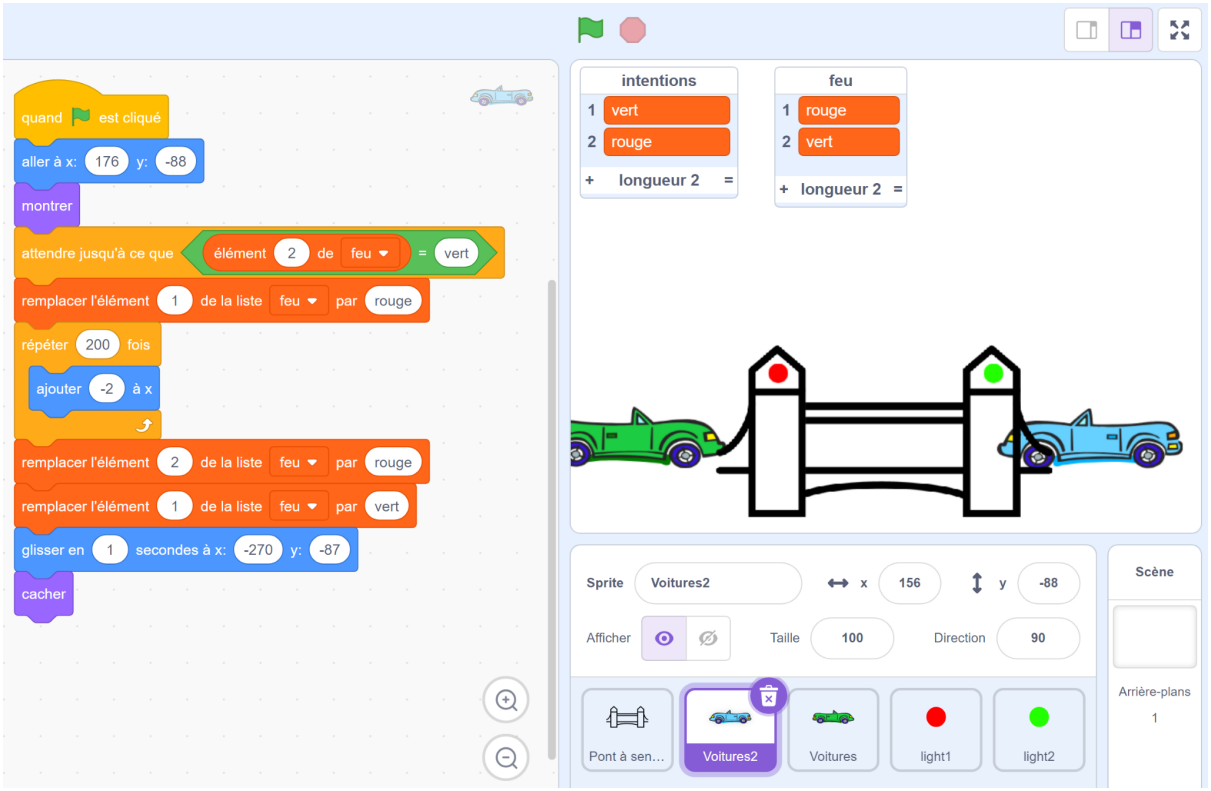


Figure 2.11 : programmation concurrente avec *Scratch*, traversée à une voie

A contrario, un objet d'apprentissage simple – tel un programme permettant à un joueur de piloter une soucoupe volante sur un écran à l'aide du clavier –, à la portée d'un enfant au stade des opérations concrètes – si cet objet est créé à l'aide d'un langage visuel tel que *Scratch* (figure 2.12) –, devient rapidement plus difficile pour un enfant de 8 ans s'il s'agit de le réaliser avec un langage de programmation littéral de haut niveau tel que *Javascript*, *Python* ou *Visual Basic*, nécessitant un plus haut degré d'abstraction (figure 2.13).

L'enseignant souhaitant initier ses élèves à l'algorithmique et à la programmation doit choisir un moyen et un objet tous deux adaptés au stade d'apprentissage de ses élèves. Si l'on observe les moyens d'apprentissage existants, la plupart d'entre eux sont conçus pour un stade d'apprentissage donné. Dès que l'enfant passe d'un stade à un autre, il est alors nécessaire de changer de moyen. Ainsi, un outil de programmation visuelle comme *Scratch* est un moyen adapté au stade des opérations concrètes. Si l'enfant passe au stade des opérations formelles dans sa découverte de la programmation, ce moyen d'apprentissage pourra lui sembler devenir trop simple (à moins que celui-ci soit utilisé sur des objets d'apprentissage plus complexe comme la programmation concurrente) et ce sera alors l'occasion de passer à un autre moyen plus formel, la plupart du temps un environnement de développement simplifié de programme dans un langage de programmation littéral, tel *Processing* ou d'autres.

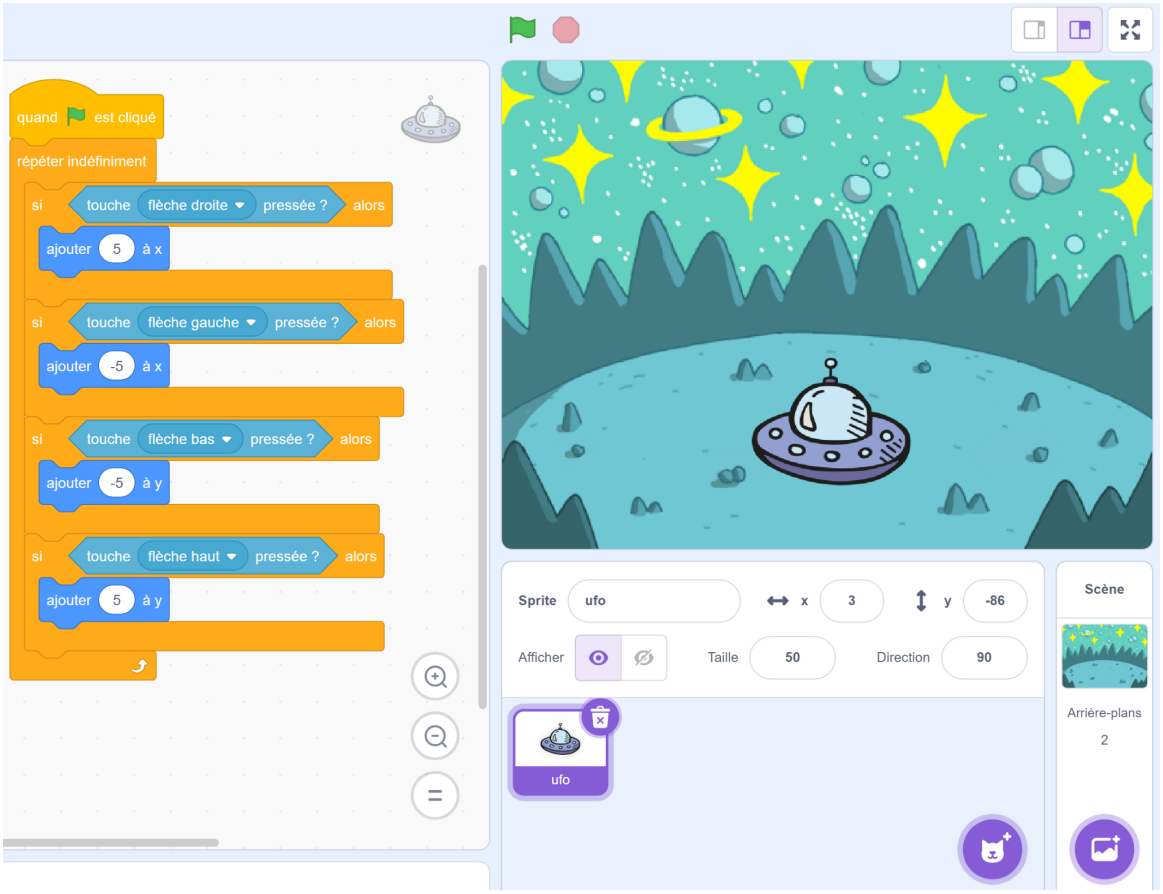


Figure 2.12 : programme de pilotage d'une soucoupe volante avec *Scratch*

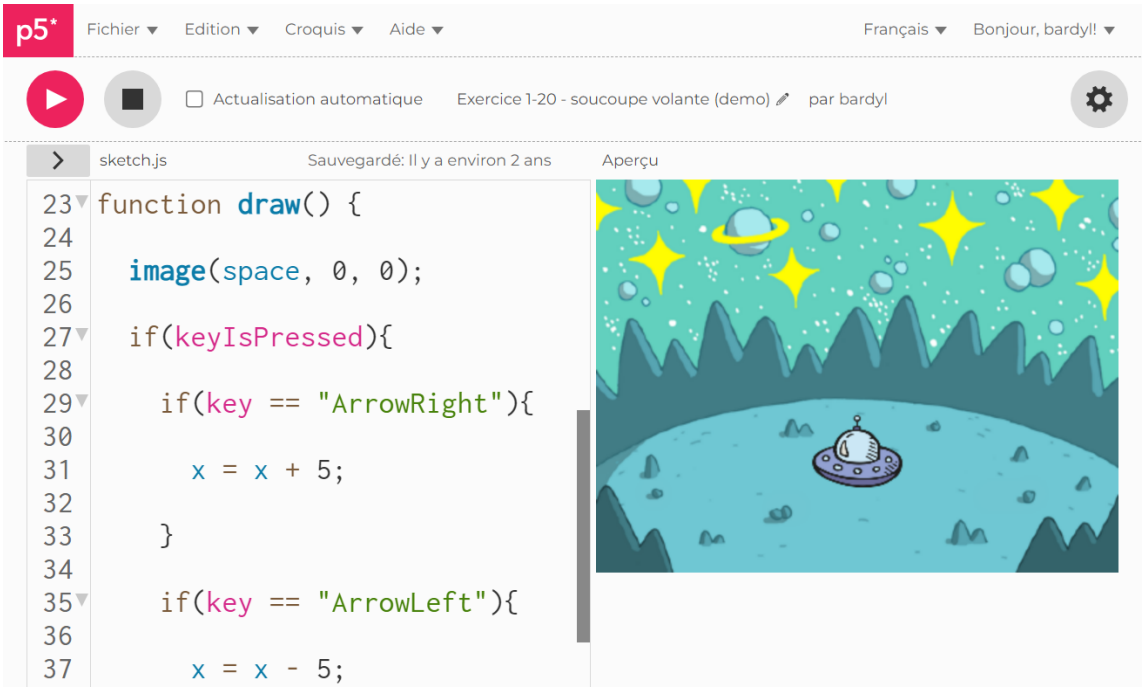


Figure 2.13 : programme de pilotage d'une soucoupe volante avec *Processing* (p5.js)

Le passage d'un stade d'apprentissage à un autre nécessite donc le plus souvent de changer d'outils d'apprentissage de la programmation. Avec les inconvénients accompagnant un tel changement de palier, à savoir un changement d'environnement impliquant un investissement en temps nécessaire à la prise en main du nouveau moyen. Un instrument intégrant deux moyens d'apprentissage distincts attachés chacun à un stade d'apprentissage différent – le second consécutif au premier – et portant sur un même objet éviterait la nécessité d'un tel changement et pourrait simplifier la transition d'un stade à un autre.

Au niveau de l'apprentissage de la programmation, un tel instrument hybride prendra la forme d'un environnement de développement simplifié permettant d'écrire un même programme à la fois, à choix, dans un langage de programmation visuel et en même temps dans un langage de programmation littéral. Cet instrument sera à même de convertir un programme écrit dans un langage visuel en un programme identique écrit dans un langage littéral.

L'environnement d'apprentissage de la programmation visuelle *Jeux Blockly* (https://blockly.games/) de Google fait un pas dans cette direction. Après l'exécution réussie d'un programme composé à l'aide de blocs visuels *Blockly* (figure 2.14), l'application affiche la version *Javascript* du programme correspondant (figure 2.15), offrant ainsi la possibilité à l'élève de découvrir comment sont représentés dans un langage de programmation littéral les blocs qu'il a utilisés.

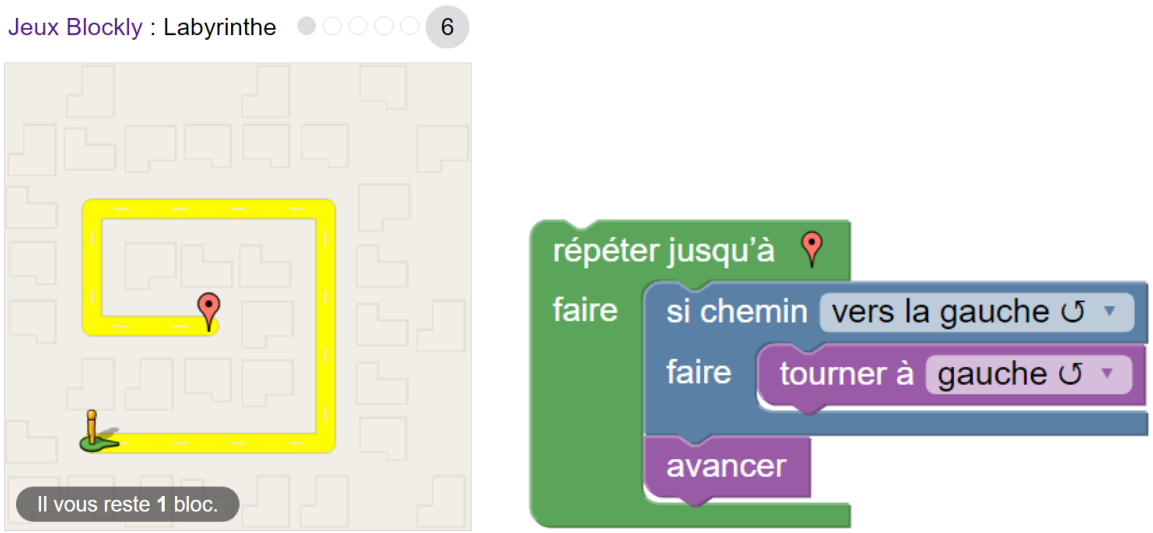


Figure 2.14 : jeux Blockly, programme visuel d'itinéraire du labyrinthe

L'application *Jeux Blockly* laisse entrapercevoir à l'enfant apprenant à programmer à partir du stade des opérations concrètes ce qui l'attend au stade suivant des opérations formelles pour la manipulation d'un même objet. Mais elle n'est pas conçue pour l'apprentissage de la la programmation avec un langage littéral au stade des opérations formelles. Elle permet de

préparer le terrain à cet effet. Afin de passer au stade suivant, il sera alors nécessaire de prendre en main un nouveau moyen d'apprentissage utilisant un langage littéral.

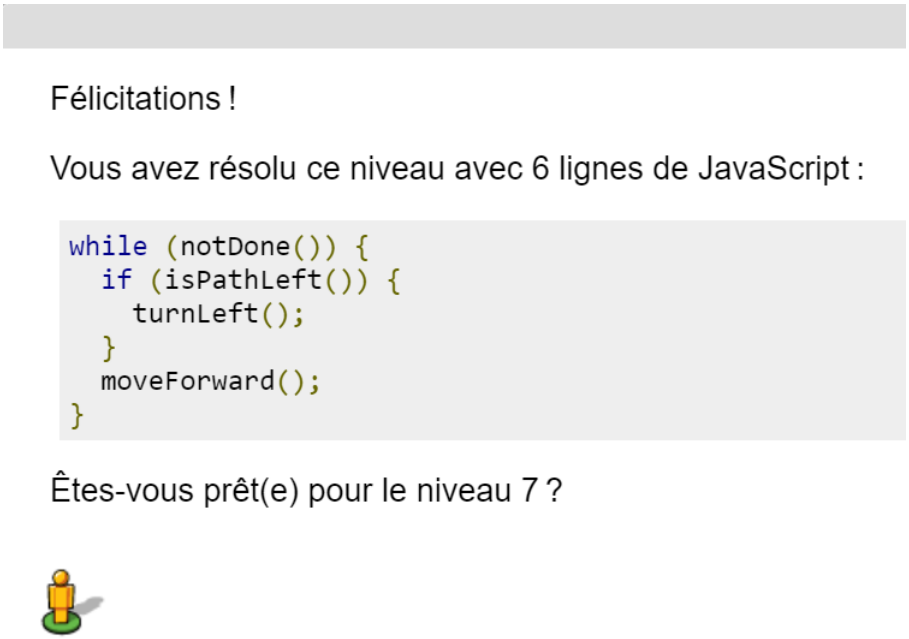


Figure 2.15 : jeux Blockly, programme littéral d'itinéraire du labyrinthe

L'application *Jeux Blockly* ne correspond donc pas pleinement à ce que nous recherchons. En revanche, le module *Microbit* de l'application *Make Code* de Microsoft (https://makecode.microbit.org/) répond à cette intention puisqu'il est possible avec celui-ci d'écrire un programme contrôlant une carte *Microbit* aussi bien dans un langage visuel (figure 2.16) que dans deux langages littéraux différents (figure 2.17ab).

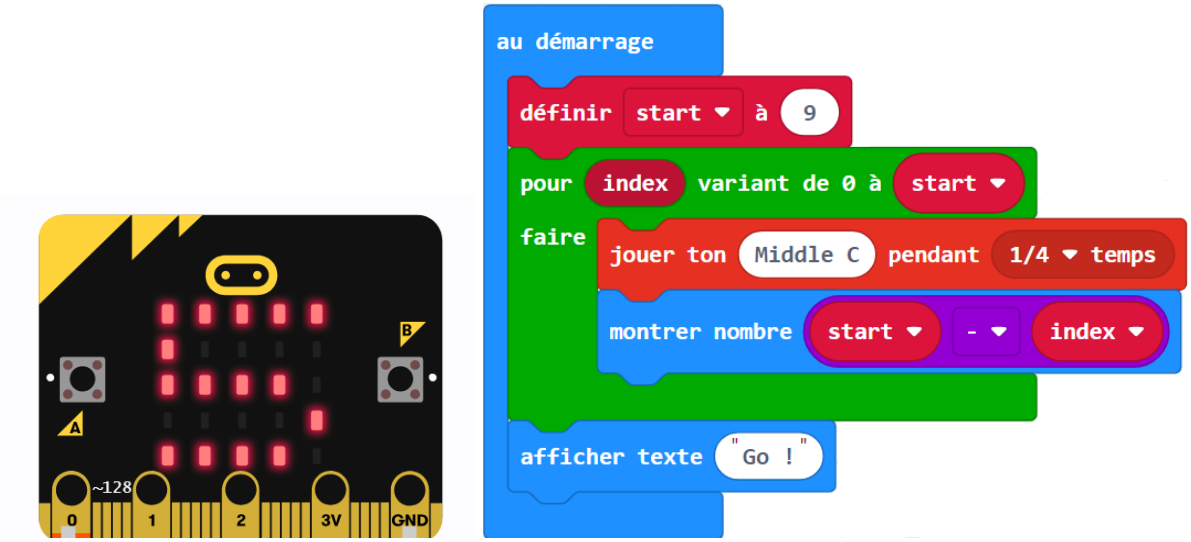


Figure 2.16 : programme visuel de compte à rebours pour *Microbit* (*Make Code*)

```
let start = 9
for (let index = 0; index <= start; index++) {
    music.playTone(262, music.beat(BeatFraction.Quarter))
    basic.showNumber(start - index)
}
basic.showString("Go !")
```

Figure 2.17a : programme *Javascript* de compte à rebours pour *Microbit* (*Make Code*)

```
start = 9
index = 0
while index <= start:
    music.play_tone(262, music.beat(BeatFraction.QUARTER))
    basic.show_number(start - index)
    index += 1
basic.show_string("Go !")
```

Figure 2.17b : programme *Python* de compte à rebours pour *Microbit* (*Make Code*)

L'éditeur du module *Microbit* de *Make Code* offre ainsi la possibilité de choisir entre trois modes d'édition différents, dont un visuel et deux littéraux (figure 2.18a). Chaque mode offre non seulement la possibilité de voir le code visuel ou littéral, mais également de le modifier et de l'éditer. A chaque passage d'un mode à un autre, le code du programme en cours d'édition est converti dans le langage du mode sélectionné. Ce qui représente un véritable défi pour les développeurs de l'application *Microbit Make Code* de par la nécessité de mettre en œuvre 2 x 3 = 6 procédures de conversion (figure 2.18b). Difficile à pleinement réussir, ces traductions peuvent susciter certaines ambiguïtés, comme on peut l'observer dans les codes des figures 2.16 et 2.17ab.

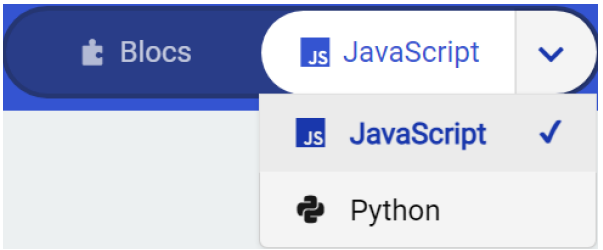


Figure 2.18a : sélecteur de mode d'édition, Microbit (*Make Code*)

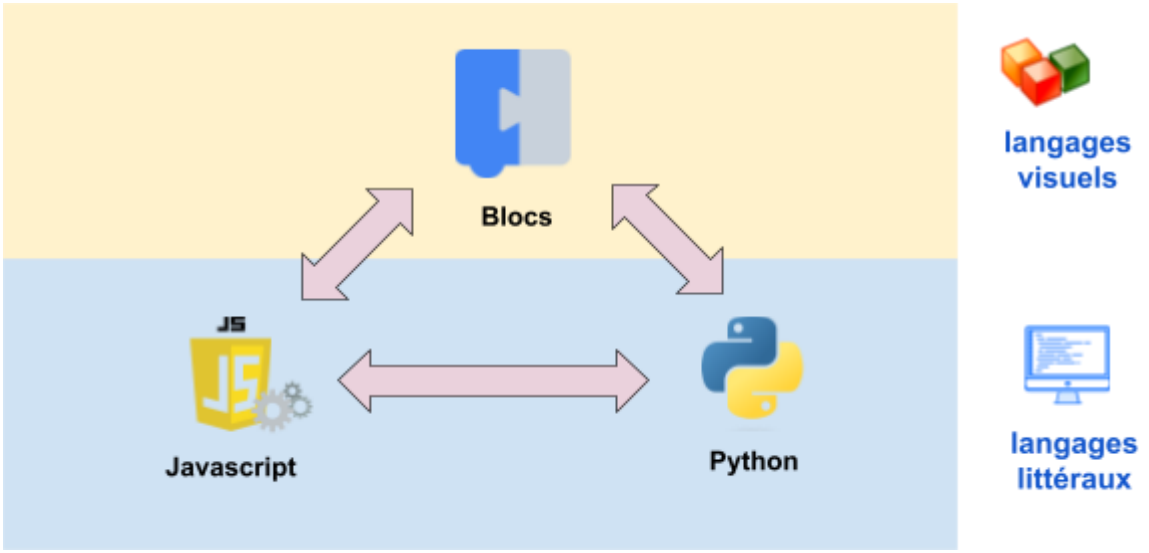


Figure 2.18b : conversions de code, Microbit (*Make Code*)

Si la structure de répétition avec compteur *pour* des blocs visuels a bien été traduite en une instruction *for*() en *Javascript*, lors de la conversion du code de *Javascript* à *Python*, l'instruction se métamorphose en un *while*() alors que l'instruction *for*() existe également dans ce second langage. Et si l'on retourne du mode *Python* au mode *Blocs*, la structure de répétition initiale *pour* se transformer en une structure *tant que*. Ces imprécisions peuvent susciter passablement de confusion chez un débutant découvrant la programmation littérale.

Ainsi qu'une certaine frustration chez l'élève ou l'étudiant souhaitant faire usage de notions et d'éléments de programmation littérale que le traducteur de l'éditeur n'est pas à même de convertir d'un langage à un autre, à l'image d'un objet. Le code de la figure 2.19 est une adaptation de celui de la figure 2.17a (compte à rebours en Javascript) dans lequel un objet a été défini (timer).

```
1  let timer = {'start' : 9, 'message' : 'Go !'}
2
3  for (let index = 0; index <= timer.start; index++) {
4      music.playTone(262, music.beat(BeatFraction.Quarter))
5      basic.showNumber(timer.start - index)
6  }
7  basic.showString(timer.message)
```

Figure 2.19 : programme *Javascript* de compte à rebours pour *Microbit* avec objet *timer*

Si l'on change de mode d'édition pour passer à l'édition en *Python*, on obtient un message d'erreur (figure 2.20) :

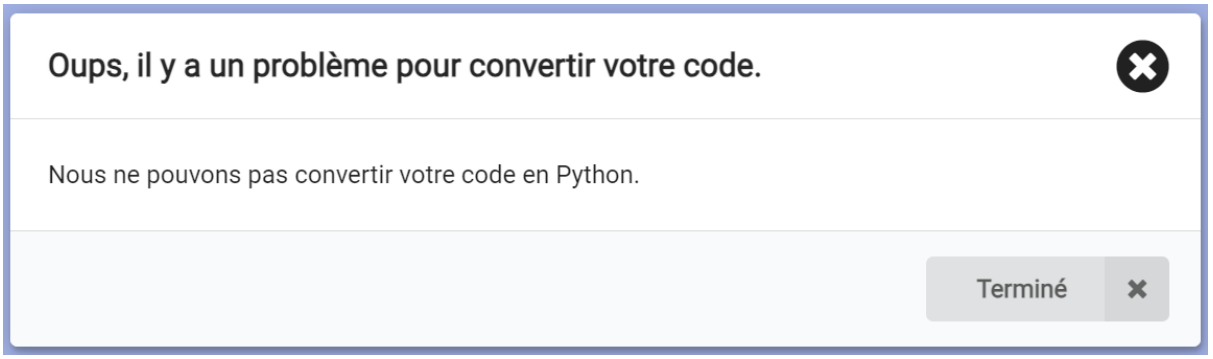


Figure 2.20 : erreur de conversion du code *Javascript* en code *Python* (*Code Maker*)

La conversion de *Javascript* en un code visuel avec blocs ne fonctionne guère mieux. Certes, aucun message d'erreur n'est produit mais le code visuel généré est inexécutable, contenant un nouveau bloc exotique représentant la définition littérale de l'objet *timer* (figure 2.21).

Figure 2.21 : erreur de conversion du code *Javascript* en code *Blocs* (*Code Maker*)

Globalement, le module *Microbit* de *Code Maker* offre une grande souplesse pour coder sous différentes formes aussi bien visuelles que littérales et permet ainsi une transition plus aisée d'un stade d'apprentissage à un autre. Cependant, ce polymorphisme tous azimuts peut dérouter un débutant de par ces imprécisions ou impossibilités de traduction. Des sens uniques et des sens interdits de traduction pourraient prévenir ces problèmes, au prix d'un degré de flexibilité moindre qui ne portera cependant pas conséquence pour un débutant.

Le passage de l'apprentissage de la programmation visuelle à celui de la programmation littérale avec *Code Maker* peut nécessiter de se plonger sans transition dans un haut niveau conceptuel, comme on peut l'observer à la ligne 3 du code *Javascript* de la figure 2.17a ou à la ligne 4 du code *Python* de la figure 2.17b. Sur une seule ligne, on trouve la méthode (*.playTone*) d'un objet (*music*), à laquelle est transmis comme second argument un objet (représentant une note de musique) instancié lui-même par une autre fonction (*beat*) d'un objet (music). C'est là un code quelque peu complexe pour un débutant : on est loin de la simplicité de *Logo*…

L'application *Sauvez Julie* est un moyen d'apprentissage de l'algorithmique et de la programmation conçu afin de découvrir cette dernière aussi bien au stade des opérations concrètes que des opérations formelles, en manipulant un objet simple et accessible à ces

stades, tout en facilitant la transition du stade inférieur au stade supérieur. L'objet d'apprentissage consiste à composer l'algorithme générant un itinéraire sécurisé à suivre par un guide pour accéder à et secourir une touriste égarée sur un glacier (figure 2.7). L'application met en parallèle deux éditeurs de code, l'un visuel et l'autre littéral (figure 2.22). Le langage de programmation visuel est constitué d'un ensemble de blocs élémentaires limités à des opérations simples (avancer, tourner à gauche, tourner à droite, sauter) et des structures de contrôle fondamentales (si … sinon, répéter *n* fois, répéter jusqu'à).

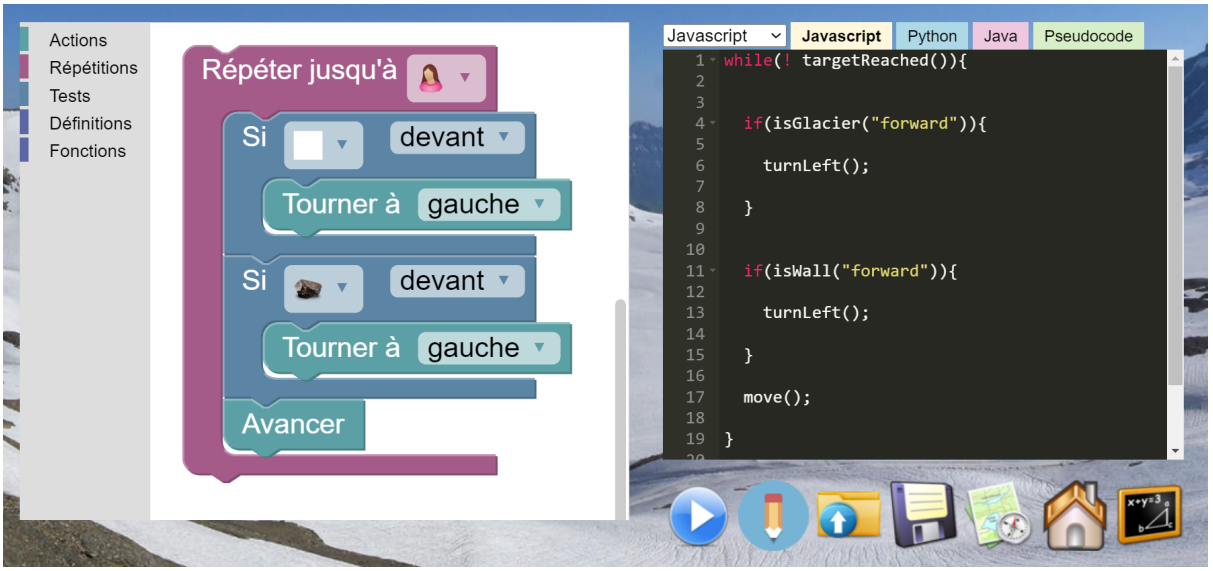


Figure 2.22 : éditeur de code visuel et littéral, *Sauvez Julie*

Le code visuel doit être converti dans un langage de programmation littéral afin de pouvoir être exécuté. Le joueur peut écrire ce code littéral par lui-même dans le second éditeur ou, comme dans *Code Maker*, faire appel à une conversion automatique. Le code littéral peut se présenter dans différents langages de programmation de haut niveau – à savoir *Javascript*, *Python* et *Java* – ainsi que sous la forme de pseudo-code.

Contrairement au module *Microbit* de *Code Maker*, le nombre et le sens de conversion est limité (figure 2.23). La conversion verticale est à sens unique, de la couche supérieure des langages visuels vers celle des langages littéraux. Un des objectifs majeurs est ici de passer d'un stade d'apprentissage inférieur à un stade d'apprentissage supérieur, soit des opérations concrètes aux opérations formelles, et non l'inverse. Une conversion des langages littéraux en un langage visuel n'est pas totalement dépourvue d'intérêt pédagogique. Elle offre notamment la possibilité de faire abstraction des langages littéraux à travers une généralisation de ceux-ci sous la forme d'un langage visuel. Cependant, cette généralisation peut se faire sans pour autant avoir besoin d'apprendre à programmer dans plusieurs langages littéraux, par simple comparaison des codes générés dans ces différents langages avec la représentation visuelle en étant la source.

Les conversions horizontales du code entre les langages littéraux n'a pas été implémentée. Celles-ci pourraient constituer un plus mais, comme nous l'avons vu avec *Code Maker*,

réalisées conjointement avec une conversion verticale remontant vers le visuel, elles sont complexes à mettre en œuvre et peuvent être source d'ambiguïté déroutante pour le débutant. La comparaison syntaxique du code d'un programme écrit dans différents langages de programmation littéraux se fait avec comme point de référence un code visuel unique. Après avoir composé son programme visuellement, l'utilisateur peut cliquer sur les différents onglets correspondant aux différents langages littéraux disponibles de l'éditeur de code afin de comparer le programme codé dans ces différents langages.

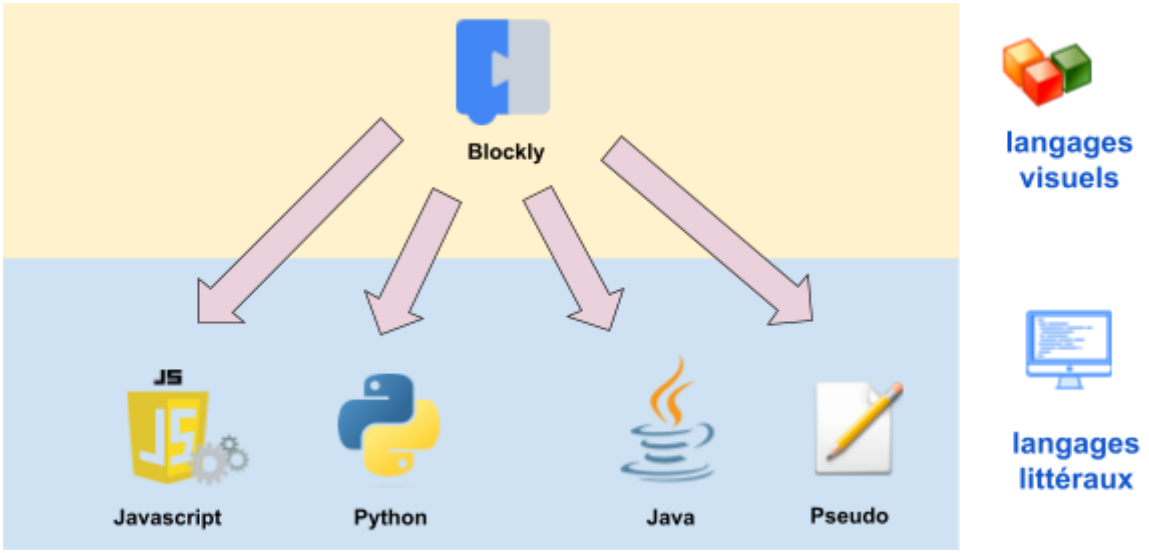


Figure 2.23 : conversions de code, Microbit (*Make Code*)

L'exécution du programme ne peut se faire qu'à partir de sa version *Javascript* car celle-ci se fait au sein d'un navigateur, dans le contexte d'une application web. Dans *Code Maker*, le code *Python* peut également être exécuté. Cependant, cela n'est possible qu'à travers une conversion cachée du code *Python* en *Javascript* et donc, au final également, seul le programme dans sa version *Javascript* est exécuté. Il serait possible d'en faire de même au sein de *Sauvez Julie*.

A cet effet, l'usage d'une bibliothèque de conversion serait nécessaire. La plupart des solutions de conversion que l'on trouvait jusqu'à il y a peu nécessitaient l'installation d'un service ad-hoc sur un serveur, auquel le code à convertir est transmis, la traduction étant ensuite retournée au client pour exécution. On peut s'étonner de telles solutions relativement lourdes. Sans doute que les développeurs de ces premières solutions souhaitaient les développer dans le langage de programmation auquel ils étaient habitués, autre que *Javascript* mais disponible côté serveur. Depuis, des solutions côté client ont vu le jour (Yegulalp S., 2023). Mais force est de constater qu'elles sont hétéroclites et s'intéressent pour l'essentiel à la conversion de *Python* vers *Javascript* ou inversément. Implémenter et assurer la maintenance l'une d'elle dans un moyen d'apprentissage de la programmation peut représenter beaucoup d'efforts pour mettre en œuvre une seule et unique conversion d'un langage littéral vers un autre.

Le développement en cours du langage de programmation de bas niveau *WebAssembly* et d'une machine virtuelle hybride dédiée à l'exécution au sein de n'importe quel navigateur web de programmes écrits dans ce langage binaire ainsi qu'en Javascript permettra de dépasser le problème du transcodage côté client web (Mozilla, 2022). Le *Webassembly* est similaire au *Common Intermediate Language* (CIL) du framework .Net de Microsoft mais au sein d'un navigateur ou d'un serveur web. Pour autant que l'on dispose d'un compilateur approprié, tout programme écrit dans un langage de haut niveau peut être compilé en *Webassembly*, cette version pouvant alors être exécutée par une machine virtuelle au sein d'un navigateur web, machine virtuelle transformant ce code intermédiaire en code machine exécuté par le processeur de l'ordinateur sur lequel le navigateur web fonctionne (figure 2.24).

Pour autant que des compilateurs correspondants soient développés à cet effet, les programmes écrits dans la plupart des langages de programmation de haut niveau les plus répandus (et même d'autres de portée moindre si des communautés se mettent à l'oeuvre à cet effet) pourront à l'avenir être exécuté dans le cadre d'un navigateur (ou d'un serveur) web sur n'importe quelle machine pour lequel le navigateur est conçu. Cela permettra à des moyens d'apprentissage web en ligne tels que *Sauvez Julie* d'offrir la possibilité d'apprendre à programmer dans la plupart des langages de programmation courant.

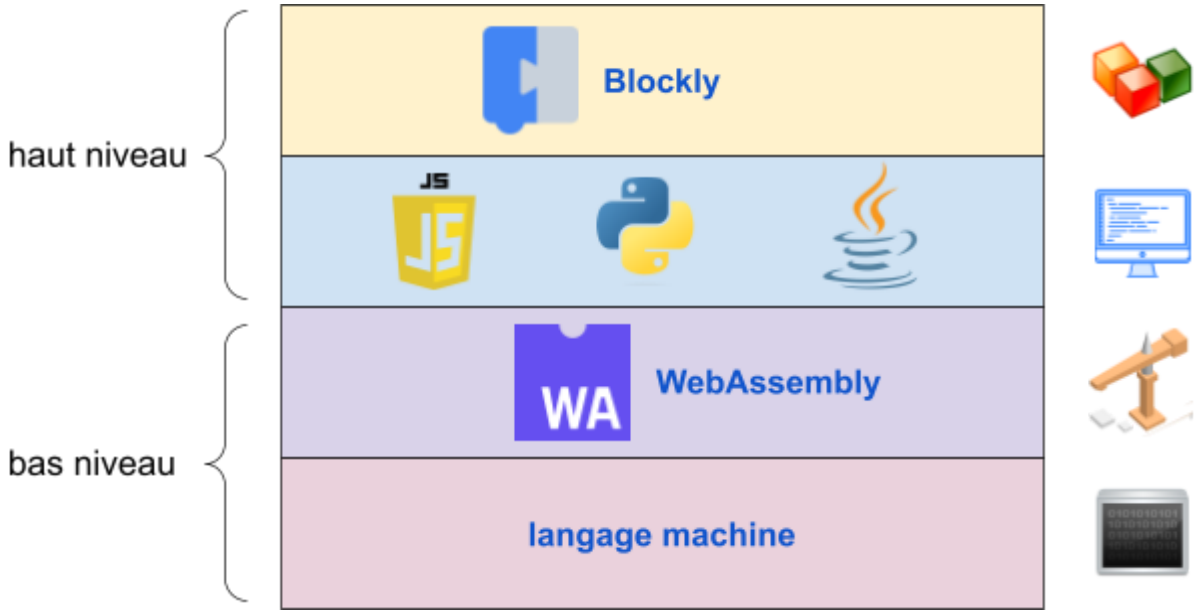


Figure 2.24 : modèle en couches des langages de programmation

2.2 Profils d'utilisateurs

En tant que moyen d'enseignement et d'apprentissage de la programmation à deux niveaux, *Sauvez Julie* peut être utilisé par différents profils d'utilisateurs, à l'image de ceux qui suivent.

a) découverte de la notion d'algorithme (Manon, 8 ans)

Manon est une élève de 5H (3ème année de l'école primaire en Suisse). Elle découvre la notion d'algorithme à l'aide de *Sauvez Julie*. Pour cela, suivant les instructions de sa maîtresse ou de son maître, elle se rend sur le site de l'application : https://apps.thike.ch/savejulie. Elle clique intuitivement sur le bouton 'Jouer' de la page d'accueil (figure 2.2.1)

Figure 2.2.1 : page d'accueil de *Sauvez Julie*

Sur instruction de son enseignant, elle clique dans la liste des exercices et tutoriels sur le premier d'entre eux, s'intitulant 'Introduction (tutoriel)' (figure 2.2.2)

Figure 2.2.2 : page de sélection des exercices et des tutoriels de *Sauvez Julie*

Après être entrée dans le tutoriel, Manon découvre grâce à une présentation de sa maîtresse ou par elle-même, en lisant les dias des présentations du tutoriel, le scénario et les consignes du jeu (figure 2.2.3). Manon revient ensuite à la liste des exercices et sélectionne le premier dont l'objet est l'apprentissage de la composition de séquences d'instructions décrivant les itinéraires que Marc, le guide, est appelé à suivre sur le glacier afin de rejoindre en toute sécurité Julie, la touriste égarée qu'il doit secourir.

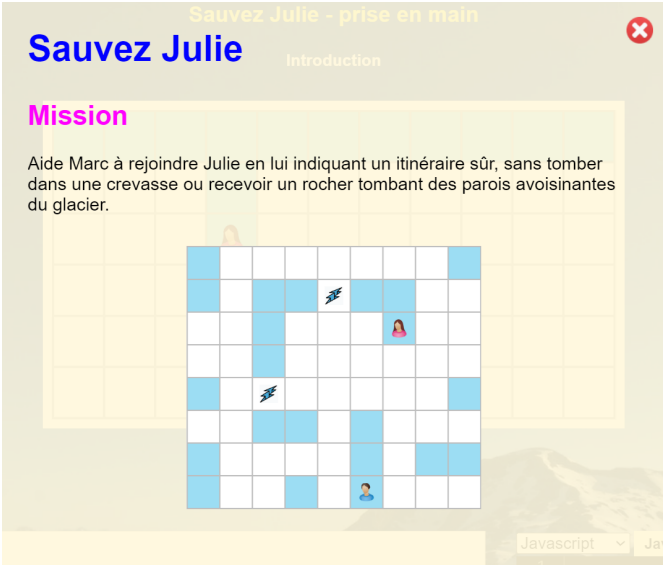


Figure 2.2.3 : tutoriel de prise en main de *Sauvez Julie*

Un exercice est constitué d'une série de cartes représentant chacune, sous la forme d'un plateau de jeu, la topographie (topo) d'un glacier ainsi que les localisations de Marc et de Julie (figure 2.2.4).

Pour chaque topo, le joueur doit composer un algorithme générant un itinéraire à suivre par le guide. Un topo peut être accompagné d'une présentation (tutoriel) constitué d'une ou de plusieurs dias. Par défaut, le tutoriel d'un topo s'ouvre automatiquement la première fois où le topo correspondant est affiché. Par la suite, le tutoriel du topo ne s'affiche plus mais le joueur peut le visionner à nouveau en cliquant sur le bouton du tableau noir (figure 2.2.5).

Figure 2.2.4 : exercice 1 - séquences d'instructions - topo No 3

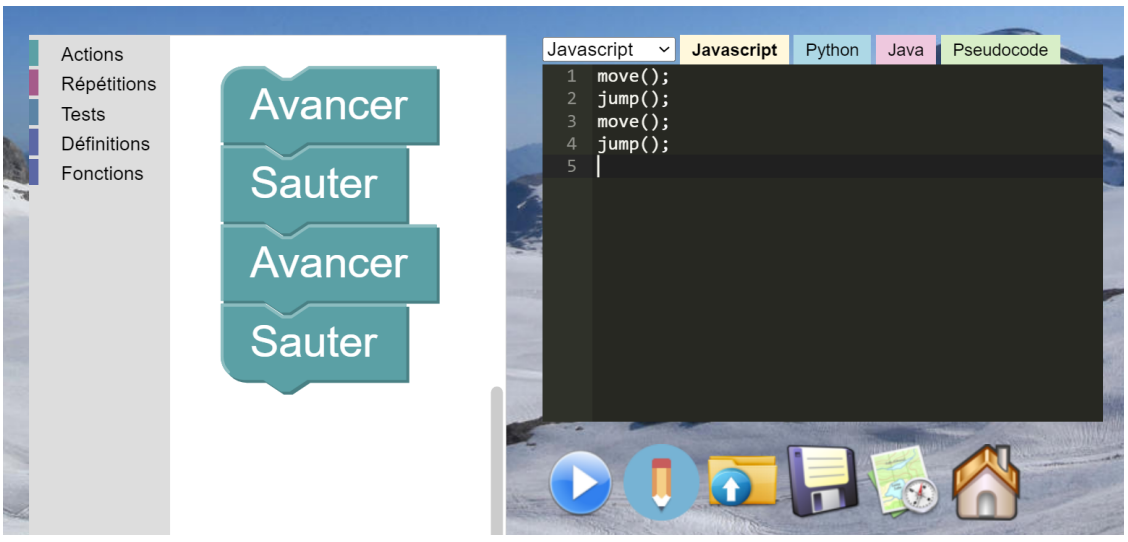


Figure 2.2.5 : exercice 1 - séquences d'instructions

Utilisant l'éditeur de blocs, Manon va concevoir un tel algorithme pour les premiers topos, à l'image du troisième de ce premier exercice (figure 2.2.5). Une fois le codage visuel de son algorithme réalisé, Manon demande à l'application de l'exécuter et de faire suivre au guide l'itinéraire généré à partir de cet algorithme. Pour cela, elle demande tout d'abord la conversion de son code visuel en code littéral en cliquant sur le bouton avec le crayon, en dessous de l'éditeur de code. Le code *Javascript* est généré et inséré dans l'éditeur.

Manon va demander l'exécution du programme qu'elle a réalisé en cliquant sur le bouton 'Jouer'. Si le guide ne parvient pas à rejoindre Julie avec cet algorithme, Manon le corrige et recommence ces opérations. En cas de succès, une séquence sonore lui indique qu'elle a réussi puis le topo suivant lui est automatiquement proposé.

Afin de ne pas avoir à demander la conversion de son code visuel en code littéral à chaque tentative d'exécution, un bouton de contrôle représentant une feuille, en haut à droite de la fenêtre de l'application, permet d'activer la conversion automatique (figure 2.2.6). A chaque fois qu'un changement est effectué dans l'éditeur de code visuel, le code littéral est à nouveau généré.

Figure 2.2.6 : boutons de contrôle

Un second bouton se présentant sous la forme d'un post-it offre la possibilité de désactiver les tutoriaux. A chaque fois que l'on ouvre un exercice contenant un tutoriel, celui-ci s'affiche à l'écran, même s'il a déjà été affiché lors d'une ouverture précédente, ce qui peut vite devenir agaçant et peut être évité à l'aide de ce bouton, qui permet également de réactiver les tutoriaux. Lorsque ceux-ci sont désactivés, il est tout de même possible d'afficher occasionnellement le tutoriel d'un topo en cliquant sur le bouton du tableau noir en-dessous de l'éditeur de code.

Enfin, le bouton avec le haut-parleur active ou désactive le fond et les effets sonores alors que les deux boutons avec la loupe changent la taille du plateau de jeu affiché.

Après avoir réussi le nombre de missions que son enseignant lui a demandé d'effectuer, Manon enregistre son exercice en cliquant sur le bouton avec la disquette, suivant les consignes de son maître, rappelant et précisant les explications données dans le tutoriel. De préférence, l'enseignant lui montre comment enregistrer l'exercice sur le lecteur web de l'école (Google Drive, OneDrive ou autre), facilement accessible depuis un ordinateur de la maison.

Manon quitte ensuite son exercice et revient à la page d'accueil puis à la page de la liste des exercices et tutoriaux (ou directement à cette dernière en cliquant sur le bouton avec la carte et la boussole). Suivant à nouveau les consignes de son enseignant, elle ouvre l'exercice qu'elle vient d'enregistrer en cliquant sur le bouton avec le dossier jaune. Elle est maintenant à même de poursuivre son exercice ou d'en commencer et d'en faire d'autres aussi bien à l'école qu'à la maison, à l'instar des exercices de base proposés par *Sauvez Julie* (figure 2.2.7), auxquels peuvent s'ajouter des exercices créés par l'enseignant.

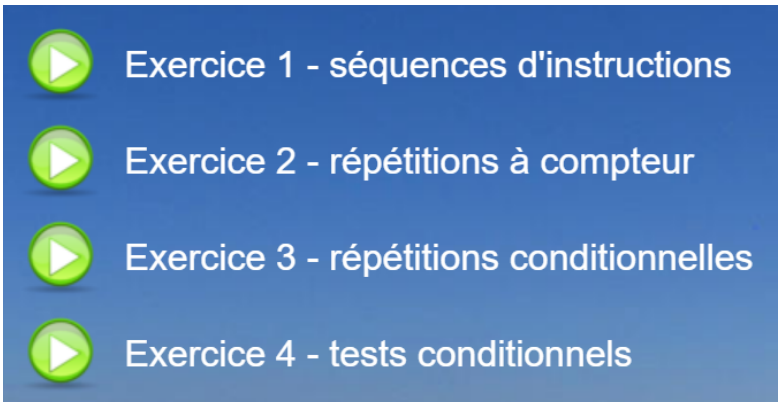


Figure 2.2.7 : exercices de base de *Sauvez Julie*

Le premier exercice permet une prise en main tout en focalisant sur l'apprentissage de la notion de séquences d'instructions élémentaires. Le deuxième offre l'opportunité de découvrir une première structure de répétition, d'un nombre de fois déterminé à l'avance, d'une séquence d'instructions. Le troisième ouvre la porte à la découverte d'une structure de répétition conditionnelle d'une séquence d'instructions lorsque le nombre de répétitions n'est pas connu au préalable alors que le quatrième se concentre sur les structures conditionnelles de branchements élémentaires et les imbrications de structures de contrôle.

Après avoir effectué les exercices de *Sauvez Julie*, Manon se dit intéressée à concevoir de petites animations avec *Scratch*, ayant observé son frère Damien réaliser un premier jeu vidéo à l'aide de cette application.

b) découverte de la programmation littérale (Damien, 12 ans)

De son côté Damien, frère aîné de Manon, termine son second semestre en classe 8H (6ème année d'école primaire), s'apprêtant à entrer en première année du cycle d'orientation (secondaire 1, classe de 9H). De par le passé, il a déjà pratiqué la programmation visuelle, en jouant avec *Sauvez Julie* ainsi que *Scratch*.

Son papa ou un enseignant lui a parlé de la possibilité d'écrire à l'aide du clavier le code de ses programmes plutôt que d'assembler visuellement des blocs. En lui disant que cela offre l'opportunité, avec un peu d'exercices, d'écrire le code d'un programme plus rapidement, tout comme de créer des programmes pouvant faire toutes sortes de tâches ainsi que des créations graphiques, des animations et des jeux plus sophistiqués, comme par exemple en 3 dimensions.

Intéressé à découvrir en quoi consiste un programme écrit dans un langage de programmation et apprenant que cela est également possible avec *Sauvez Julie*, Damien ouvre l'application et, dans un premier temps, refait à l'aide de blocs visuels les algorithmes des premiers topos de l'exercice consacré aux séquences d'instructions. Après chaque composition d'un algorithme, il en demande la conversion en *Javascript* et observe le code obtenu. Puis le défi suivant lui est lancé : écrire par lui-même le code de chaque topo en Javascript. Ce qu'il parvient aisément à réaliser (figure 2.2.8).

Figure 2.2.8 : code *Javascript* écrit par Damien pour le topo No 3 de l'exercice 1

Damien va procéder quelque peu différemment avec les exercices sur les structures de contrôle et les imbrications, à l'instar du second exercice consacré aux répétitions avec compteur. Il bénéficie d'une explication du rôle, de la syntaxe et des paramètres de l'instruction *for*() sous forme de démonstrations dans l'éditeur en ligne de l'application *p5js*

(https://editor.p5js.org/) avec de petits programmes écrivant des séries de nombre ou de texte dans la console *Javascript* (figure 2.2.9).

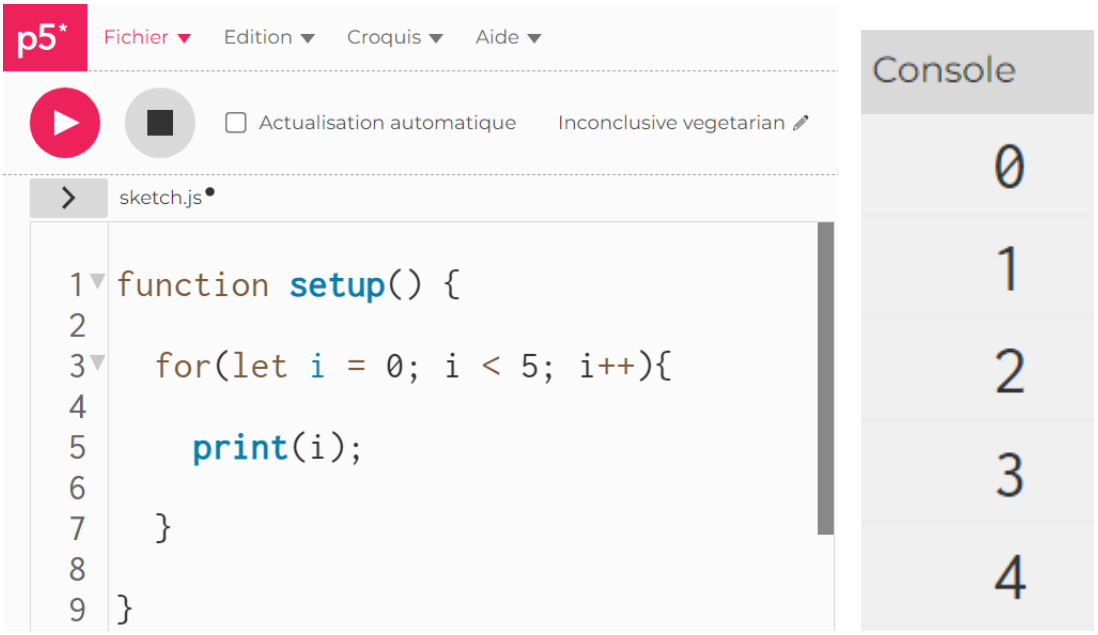


Figure 2.2.9 : découverte de l'instruction *for*() de *Javascript*

Damien va ensuite observer plus en détail les codes *Javascript* obtenus à partir de la conversion de programmes visuels conçus pour quelques topos, puis faire des essais de conversion de blocs de structures de contrôle élémentaires pour en observer le résultat. Avant de se lancer dans ses premières tentatives de rédaction de code, il copie/colle dans un bloc-note des codes *Javascript* de structures de contrôles élémentaires obtenus lors de ses expérimentations. Puis il utilise ces extraits de code en les copiant / collant en sens inverse dans l'éditeur de code de *Sauvez Julie* pour la rédaction d'un topo spécifique. Il reprend par exemple ainsi le code d'un bloc 'Répéter 3 fois', l'adapte pour obtenir 5 répétitions et place dans les accolades la séquence d'instructions élémentaires à répéter (figure 2.2.10).

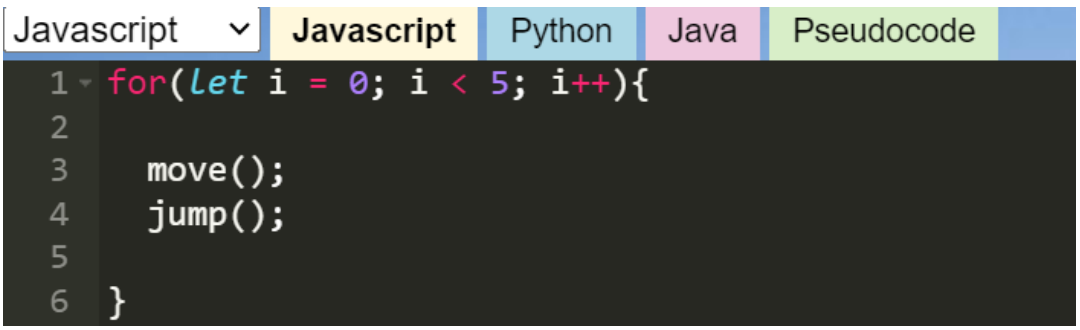


Figure 2.2.10 : code *Javascript* écrit par Damien pour le topo No 3 de l'exercice 2

Damien rencontre deux principales difficultés pour lesquelles il a parfois, dans un premier temps, besoin d'aide. La première réside dans la précision des sélections de code à copier et l'oubli d'une accolade de fermeture qui pose ensuite un problème de syntaxe lors de l'insertion de ce code dans le code final à compléter. Petit à petit cependant, Damien commence par lui-même à corriger en aval son erreur en ajoutant l'accolade manquante dans le code final. Seconde difficulté, sans incidence sur le fonctionnement du programme écrit en Javascript, l'oubli d'indentation aux instructions imbriquées dans une structure de contrôle. Il est fréquemment nécessaire à l'enseignant de rappeler la nécessité de cette indentation afin d'améliorer la lisibilité de la structure du code, en attirant l'attention sur l'indentation naturelle des blocs visuels imbriqués. Mais l'habitude se prend petit à petit, même si l'enthousiasme de vouloir rapidement tester une solution peut l'emporter sur le souci de finition du code.

Damien découvre l'instruction *if*() et les imbrications de structures de contrôle dans l'exercice 4 après une présentation de la syntaxe et du fonctionnement de cette instruction, avant de composer le code *Javascript* pour le premier topo, à nouveau en copiant / collant le code de structures de contrôles (figure 2.2.11). Il aura cependant besoin de quelques indices pour saisir l'intérêt d'une imbrication. Dans un premier temps, Damien compose le code des lignes 4 à 12 (*if*), le teste et est surpris que Marc n'atteigne pas Julie mais ne fasse qu'un seul déplacement. Avec la prise de conscience de la nécessité de répéter l'exécution de cette structure *if*() jusqu'à ce que Marc rejoigne Julie, Damien imbrique alors spontanément cette structure de branchement dans une structure de répétition *while*().

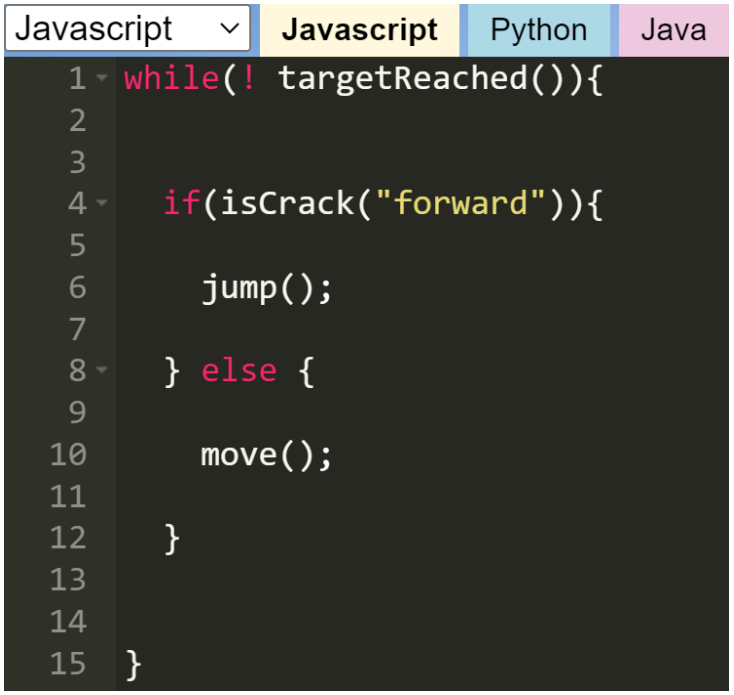


Figure 2.2.11 : code Javascript écrit par Damien pour le topo No 1 de l'exercice 4

Pour le topo No 4 de l'exercice 4 (figure 2.2.12), Damien aboutit à un code inédit (figure 2.2.14) qui n'était pas celui que l'auteur d'un problème pour débutant avait en tête, à savoir une imbrication de structure de premier niveau (figure 2.2.13).

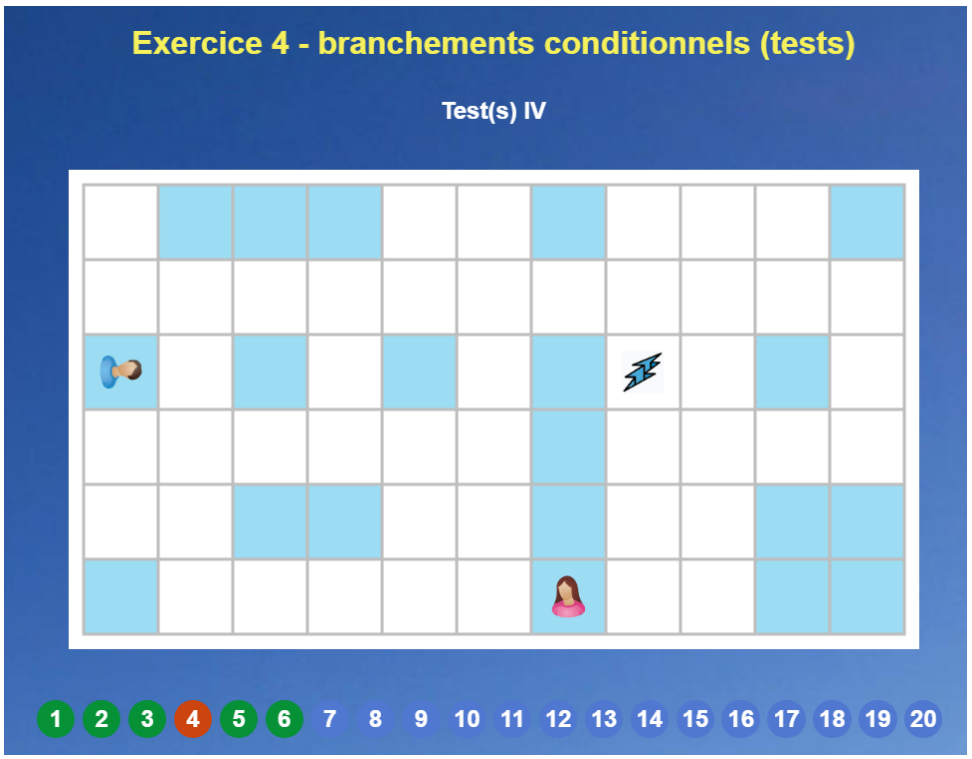


Figure 2.2.12 : exercice 4 - branchements et imbrications - topo No 4

Javascript | Javascript | Python | Java | Pseudocode

```javascript
while(! targetReached()){

  if(isGlacier("forward")){

    jump();

  }

  if(isCrack("forward")){

     turnRight();

  }


  if(isTrack("forward")){

     move();

  }

}
```

Figure 2.2.13 : code *Javascript* attendu pour le topo No 4 de l'exercice 4

Cependant, spontanément et malgré des conseils l'incitant à aller en direction de la solution escomptée, Damien préfère opter pour une imbrication à deux niveaux d'un branchement alternatif dans un autre branchement alternatif lui-même placé dans une répétition (figure 2.2.14). Cette solution n'était pas du tout attendue car jugée trop complexe pour un débutant sur le plan du codage formel. Si l'intuition était appropriée et la solution fonctionnelle, après ajout d'une accolade manquante, il a tout de même encore fallu insister sur la nécessité des indentations pour la lisibilité du code.

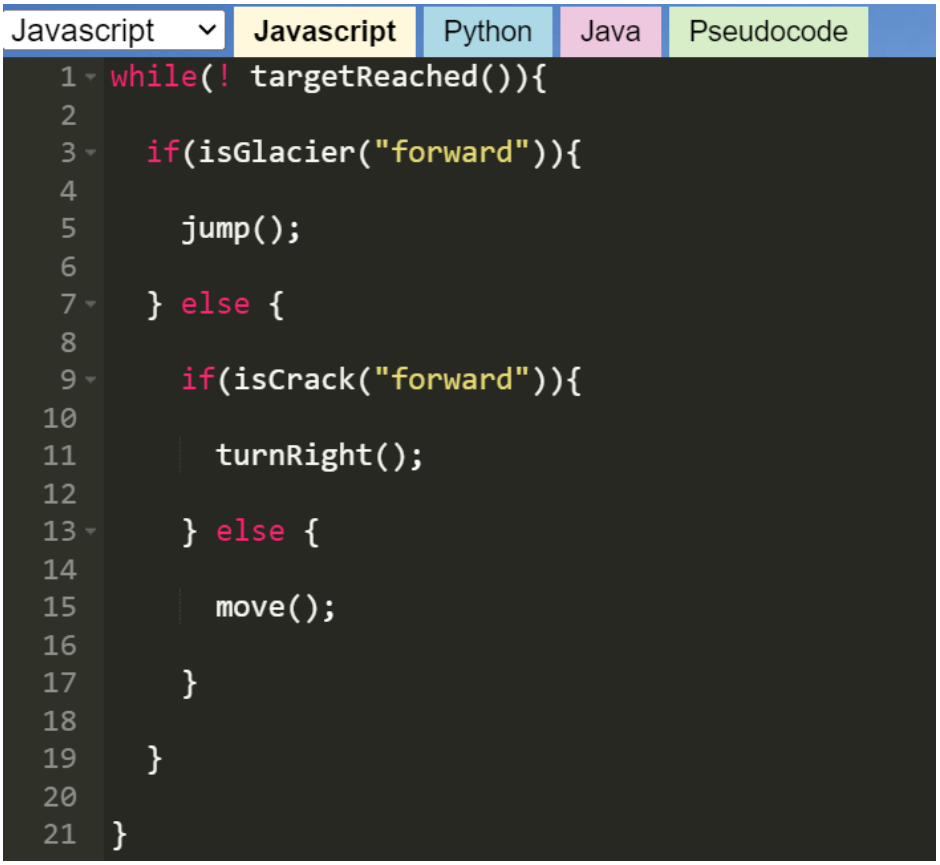


Figure 2.2.14 : code Javascript écrit par Damien pour le topo No 4 de l'exercice 4

A ce stade, Damien n'a pas appris à rédiger un tel code par lui-même du début à la fin. Cet apprentissage nécessiterait sensiblement plus de temps de pratique et d'exercices que celui se basant sur des copier / coller de code à adapter. C'était cette seconde approche, d'apprentissage classique du codage ex-nihilo dans un éditeur vierge, que le concepteur de *Sauvez Julie* avait en tête, pour ce second stade de découverte de la programmation. Force est de constater que Damien a préféré passer par une étape préalable d'apprentissage par imitation et adaptation.

Cela interroge sur la meilleure approche pour aborder la programmation littérale : faut-il apprendre à écrire dans un langage de programmation à partir de définitions ou directement à l'aide d'exemple de code (voir également ci-dessous) ? La mise à disposition de l'élève d'un formulaire prenant la forme d'un mémento des notions élémentaires illustrées par des exemples simples ne simplifierait-elle pas l'apprentissage de la programmation, de la même

manière que le 'Formulaires et tables de Mathématiques, Physique et Chimie' (CRM, 2018) facilite l'apprentissage de ces disciplines ?

Damien a pesté à chaque fois que son code était effacé suite à un manipulation effectuée dans l'éditeur visuel, sa petite sœur ayant laissé active la conversion automatique du code visuel en code littéral, avant que Damien ne désactive cet automatisme. Damien pourrait encore par la suite, après avoir assimilé les notions élémentaires de programmation exercées dans *Sauvez Julie*, comparer le code *Javascript* obtenu avec d'autres versions écrites dans d'autres langages tels que *Python* et *Java*. A partir de là, il serait possible de lui proposer de transposer les notions assimilées d'un langage à un autre, en lui demandant de convertir par lui-même le code littéral d'un programme donné dans un autre langage de programmation, lui permettant ainsi de généraliser le concept de langage de programmation et de passer à un niveau d'abstraction supérieur.

Aussi bien dans l'apprentissage visuel que dans l'apprentissage littéral, Manon et Damien ont à plusieurs reprises dû interpréter leur code lorsque celui-ci comportait une erreur. En l'exécutant manuellement, pas à pas, sur l'écran afin de repérer leur erreur. Un débogueur permettant une telle exécution pas à pas serait un plus, mais compte-tenu de la simplicité de l'objet d'apprentissage, l'expérience a montré qu'à ce niveau il est possible de s'en passer.

Cette manière de procéder soulève une seconde question intéressante : quelle place accorder à l'interprétation de code par un débutant, sans nécessairement lui fournir des explications préalables ? Un cours ordinaire suit le plus souvent une approche déductive partant de l'abstrait pour aller vers le spécifique (présentation générale, exemples, exercices pratiques). Une approche déductive en sens inverse pourrait-elle être plus productive : apprentissage par l'observation suivie d'une généralisation, explicite ou implicite ? C'est l'approche pour laquelle a par exemple opté l'équipe française de Toxicode avec les moyens d'apprentissage en ligne *Compute it* (https://compute-it.toxicode.fr/) (figure 2.2.15) et *Silent Teacher* (https://silentteacher.toxicode.fr/) (figure 2.2.16).

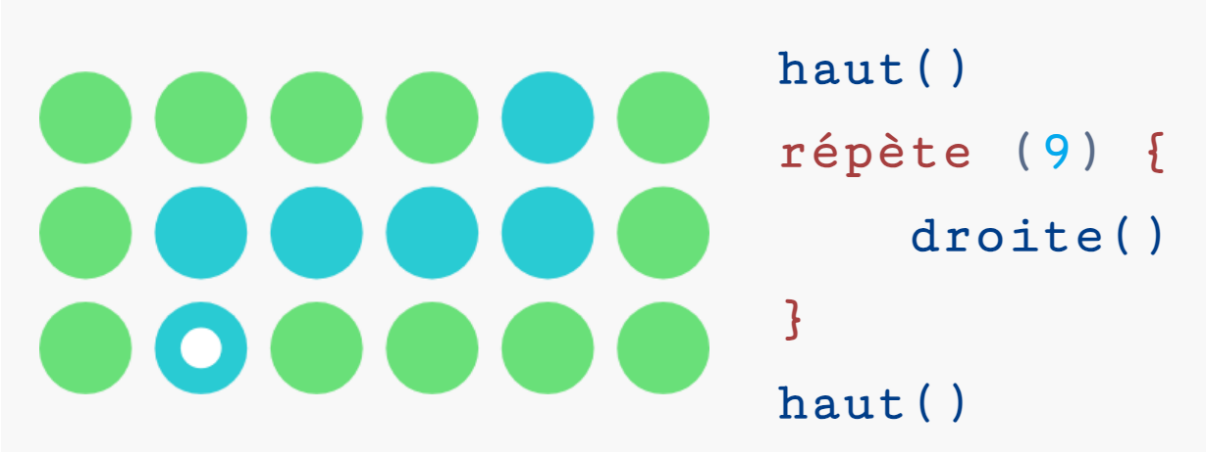


Figure 2.2.15 : application web d'apprentissage de la programmation *Compute it*

La question reste ouverte, mais il n'empêche qu'on observe une certaine contradiction quand on compare la manière dont la programmation est souvent enseignée de manière conventionnelle dans une salle de classe et la pratique d'apprentissage des informaticiens lorsqu'ils découvrent un nouveau langage ou une nouvelle technologie, préférant souvent opter pour une approche pratique par mimétisme, adaptation et essais et erreurs, notamment en recopiant, adaptant et assemblant des parties de code, comme l'a fait spontanément Damien… C'est là finalement l'approche pédagogique développée par Piaget et basé sur les observations de la manière dont les enfants apprennent : en manipulant des objets physiques et virtuels dont ils découvrent les attributs et les relations qu'ils entretiennent.

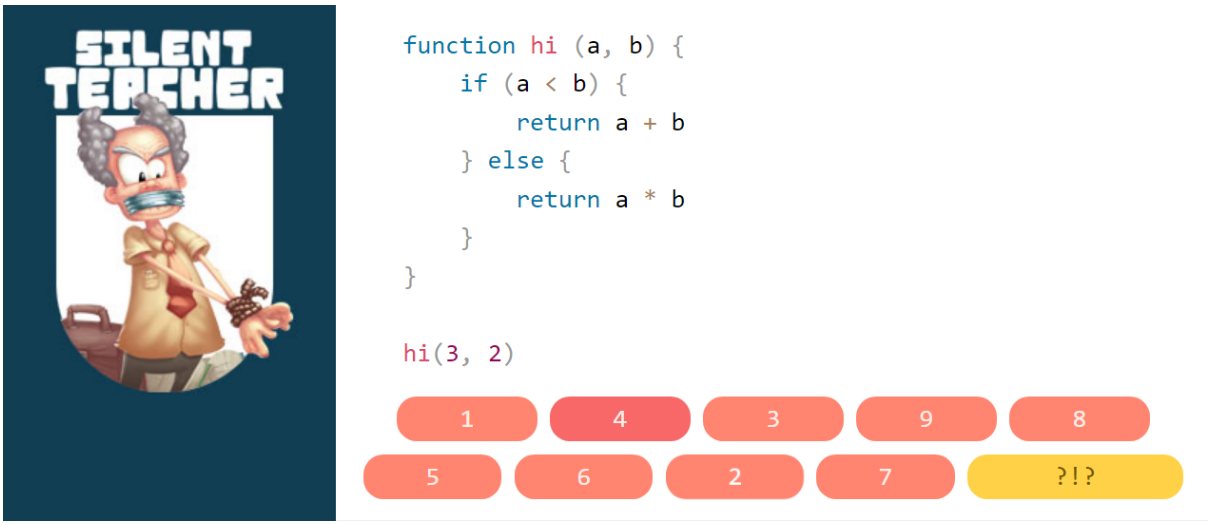


Figure 2.2.16 : application web d'apprentissage de la programmation *Silent Teacher*

c) préparation d'exercices (Laurent, enseignant, 52 ans)

Laurent donne un cours d'introduction à l'informatique dans des classes de première année d'un gymnase. Convaincu par des collègues qu'aborder initialement la programmation à l'aide d'un moyen d'apprentissage visuel est une approche efficace, même avec des élèves de 15-16 ans en étant déjà au stade de l'apprentissage des manipulations formelles, il utilise *Sauvez Julie* avec ses élèves durant les 3-4 premières leçons du chapitre de programmation. Cela prépare le terrain pour l'apprentissage de la programmation dans un langage de programmation littéral. Après ces premières leçons, il passe à *Scratch* comme second environnement visuel de programmation permettant à ses élèves d'aller plus loin dans la découverte des notions de programmation, en manipulant un objet d'apprentissage plus élaboré. Ses élèves y développent de petits jeux vidéos sous forme d'exercices, à l'aide de tutoriaux, et peuvent ensuite se lancer dans un projet de conception de leur propre jeu.

Une fois les principales notions élémentaires de programmation étudiées, celles-ci sont alors transposées au niveau formel dans un langage de programmation. Pour cela, Laurent a à sa disposition différentes plateformes d'apprentissage de la programmation littérales en ligne telles que l'éditeur p5js de *Processing*, les modules de *Code Maker* ou d'autres moyens.

Mais il peut également tirer profit de la connaissance qu'ont déjà ses élèves de *Sauvez Julie* afin de découvrir dans un environnement simple et épuré les rudiments de la programmation littérale, qu'ils pourront ensuite approfondir avec les autres moyens cités précédemment.

Pour cela, Laurent peut utiliser les exercices de base mis à disposition par *Sauvez Julie* mais également concevoir ses propres exercices et évaluations avec l'outil de création d'exercices et d'édition de topos (figure 2.2.17). Afin d'y accéder, il lui suffit de cliquer sur le bouton avec la clé anglaise de la page d'accueil de l'application.

Figure 2.2.17 : outil de création d'exercices et d'édition de topos de *Sauvez Julie*

Laurent peut également ajouter un tutoriel à un exercice. Pour cela, il lui est nécessaire de générer une arborescence de dossiers sur le site web de *Sauvez Julie*, d'y créer un fichier *.json* contenant les différentes présentations et dias du tutoriel ainsi que de placer dans les dossiers appropriés les fichiers constituant les ressources multimédia (images, vidéos) du tutoriel. Pour cela, il procédera comme Damien en créant un nouveau tutoriel par analogie à un tutoriel déjà existant, qu'il pourra copier / coller et adapter selon ses besoins. Et afin de comprendre plus aisément la structure des fichiers et dossiers constituant un tutoriel, il

pourra consulter la deuxième partie de ce document, à l'aide de laquelle il pourra également adapter et poursuivre le développement de *Sauvez Julie* à travers la compréhension de ses principaux mécanismes constitutifs.

3. Eléments de programmation

Sauvez Julie est développée avec des technologies et une architecture web et est composée d'une interface élémentaire de programmation permettant de générer le code à exécuter par un moteur d'animation. Cette troisième partie présente quelques éléments fondamentaux de l'application.

3.1 Technologies et architecture

L'application repose fondamentalement sur la technologie web côté client, c'est-à-dire celle que l'on trouve au sein d'un navigateur web, qu'elle utilise comme interface utilisateur ainsi que comme moteur d'exécution. L'application a elle-même été intégralement développée au sein de l'IDE d'un navigateur, à savoir de celui de *Chrome*, qui intègre un multitude d'outils à cet effet, à commencer par un éditeur et un débogueur mais également des outils d'analyse du trafic de données échangées avec les serveurs ou de gestion de la mémoire des applications web.

a) technologies

L'interface de l'application est constituée exclusivement de pages web faisant ainsi appel au langages ***HTML*** et ***CSS*** ainsi qu'aux interpréteurs du navigateur web générant visuellement ces pages à partir de leurs codes. Le code source de l'application est intégralement développé en *Javascript* et intégré dans ses pages web. Le framework ***Processing*** dans sa version ***Javascript*** est ajouté comme couche supérieure de ce langage, sous la forme d'une bibliothèque d'extensions en charge de la création et de l'animation graphique. Le framework ***JQuery*** est également sollicité pour simplifier quelques manipulations d'éléments *HTML* et *CSS*. La bibliothèque ***Blockly*** est utilisée afin d'offrir à l'utilisateur un éditeur de code visuel composé d'un assemblage de blocs. Enfin, l'application web ***Ace*** est intégrée comme membre de l'application principale en tant qu'éditeur de code source (figure 3.1.1)

Les langages *HTML* et *CSS* sont des langages déclaratifs décrivant le contenu et la mise en forme d'une page web, le navigateur web se chargeant d'afficher la page décrite. Ces deux langages sont utilisés dans *Sauvez Julie* afin de définir l'interface utilisateur graphique de l'application. *Javascript* est un langage impératif gérant notamment les interactions de l'utilisateur avec l'interface web alors que *Processing* est un framework multilingue multimédia permettant de faire des dessins en 2 ou 3 dimensions sur un canevas, de réaliser des animations en 2d ou 3d ainsi que de traiter des données multimédias. Ce framework est disponible en *Java*, *Javascript* et *Python* dans sa version desktop ainsi qu'en *Javascript* dans sa variante web s'exécutant dans un navigateur web.

JQuery est un framework constitué d'une bibliothèque *Javascript* contenant des fonctions simplifiant la manipulations des éléments *HTML* d'une page web hiérarchisés dans un arbre (DOM) ainsi que leurs propriétés *CSS*, les interactions de l'utilisateur avec la page tout comme ses transferts de données de et vers un serveur. *Sauvez Julie* se cantonne à utiliser des fonctions manipulant les valeurs de propriétés CSS des éléments d'une page.

Ace est un éditeur de code source prenant la forme d'une application *Javascript* pouvant être intégrée dans une page ou une application web. Il s'agit d'un éditeur multilangages avec coloriage et mise en évidence la structure du code et les erreurs de syntaxe, intégrant également de la complétion de code.

Enfin, *Blocky* est une bibliothèque *Javascript* développée par Google offrant la possibilité d'intégrer dans une page web un éditeur de code visuel composé de blocs représentant les éléments fondamentaux d'un langage informatique, qu'il s'agisse d'un langage de programmation ou d'un langage déclaratif tel que le *HTML*. Cette bibliothèque met également à disposition des fonctions traduisant le code visuel en code source d'un langage donné. *Blockly* offre des blocs de base par défaut, mais il est cependant possible de définir de nouveaux blocs spécifiques. Il permet également de générer un espace de travail personnalisé, à savoir un éditeur de code visuel s'intégrant dans une page web.

Figure 3.1.1 : modèle en couche des technologies de l'application *Sauvez Julie*

Ces technologies mettent en œuvre les différents éléments de l'architecture de *Sauvez Julie*.

b) architecture

L'application est composée de pages *HTML* (figure 3.1.3), les deux premières étant des pages statiques (dont le contenu n'évolue pas) jouant le rôle d'interface utilisateur graphique permettant d'accéder à ses différentes ressources et modules. La page d'accueil (*index.html*, figure 2.2.1) renvoie l'utilisateur vers les les pages dynamiques de programme (éditeur de topo, plateau de jeu) ainsi qu'à la page des topos (*topos.html*) proposant des ressources du cours (exercices, documents, figure 2.2.2).

L'application est également composée de pages *HTML* dynamiques : leur contenu évolue en fonction des actions de l'utilisateur de l'application interagissant avec ses composants. Ces pages constituent les deux principaux modules de l'application : l'éditeur de topos (*editor.html*, figure 2.2.17) et le plateau de jeu avec ses éditeurs de code (*play.html*, figure 2.2.4 et 2.2.5). Le premier est utilisé afin de créer de nouveaux exercices. Un exercice, constitué d'une série de topos, est affiché sur le plateau de jeu. Le joueur y édite les itinéraires qu'il demande au guide de suivre sur les différents topos lui étant soumis. Le plateau de jeu est ainsi composé d'une interface de programmation avec laquelle les algorithmes générant les itinéraires à suivre par le guide sont formulés dans des éditeurs. On y trouve également une surface de jeu sur laquelle le topo est affiché. Le module comprend un moteur d'animation mettant en oeuvre sur ce topo les mouvements du guide suivant l'itinéraire que le joueur lui a composé.

Figure 3.1.2 : tutoriel - dia d'une présentation dans *Sauvez Julie*

Le plateau de jeu dispose d'un module d'extension optionnel. Celui-ci se présente sous la forme d'un *tutoriel* pouvant s'afficher en surimpression par dessus le plateau jeu. Chaque topo peut être accompagné d'une présentation constituée d'une ou plusieurs dias que le joueur peut faire défiler séquentiellement. Chaque dia est composée d'un titre, d'un thème auxquels peuvent s'ajouter une explication ainsi qu'une image et une vidéo (figure 3.1.2). Le plateau de jeu affiche les présentations du tutoriel si celui-ci existe dans l'exercice affiché et est activé.

Figure 3.1.3 : modules de l'application *Sauvez Julie*

La conservation des données d'exercices de l'utilisateur de l'application se fait au sein de fichiers dans le format *.json* (figure 3.1.4). L'utilisateur télécharge sur un support de stockage l'exercice qu'il a tout ou partie réalisé afin de pouvoir le poursuivre ou le consulter ultérieurement. Il en fait de même avec les nouveaux exercices vierges qu'il conçoit par le biais de l'éditeur de topos. Ceux-ci peuvent être manuellement publiés en intranet ou sur Internet par un administrateur de site web de l'application, dans la page des topos afin d'en faciliter l'accès. On y trouve aussi une série d'exercices de base, également stockés sur le site web de l'application.

Figure 3.1.4 : stockage externe des données (fichiers)

Les données de l'application devant être conservées d'un module (page web) à un autre – lors d'un rechargement du module – ou d'une session à un autre de l'utilisateur sont placées dans l'espace de stockage local du navigateur web. Elles ne peuvent être mises dans des variables superglobales, partagées par différentes pages web, car ce type de variable n'existe pas en *Javascript* pour des raisons de sécurité. Dans le cas d'un ordinateur utilisé par différents utilisateurs ayant chacun leur compte, le navigateur leur attribue à chacun un espace de stockage personnel, préservant ainsi la confidentialité des données y étant placées. Cet espace de stockage est cloisonné par site ou application web, de telle sorte à ce qu'une application web n'ait pas accès aux données de l'utilisateur d'une autre application web ouverte dans le même navigateur.

Sauvez Julie mémorise deux catégories d'informations dans cet espace : d'une part des paramètres du plateau de jeu (variables d'état déterminant l'activation ou non de l'audio, de l'auto-génération du code source et de l'affichage du tutoriel), et d'autre part l'ensemble des topos de l'exercice (univers) en cours. Ces derniers sont mémorisés dans une variable globale du module *play.html*. Cependant, lorsque la page doit se recharger à nouveau dans le navigateur, ces données sont perdues. C'est ce qui se passe lors de l'ouverture par l'utilisateur d'un fichier local contenant les données d'un exercice, le fichier choisi avec la boîte de dialogue 'Fichier → Ouvrir…' devant être transmis à une nouvelle page web pour traitement de son contenu. Ce fichier étant ouvert depuis la page *play.html* et son contenu devant également être traité par celle-ci, elle doit ainsi se recharger à cet effet.

Les paramètres utilisateur de l'application étant stockés dans son espace de stockage local du navigateur (figure 3.1.5), leurs valeurs ne sont pas perdues d'une session à une autre, au

moment où l'utilisateur ferme son navigateur (contrairement aux données stockées dans l'espace de session).

Figure 3.1.5 : stockage interne des données (stockage local du navigateur)

Le dossier 'topos' contient les fichiers d'exercices. Le code source de l'application ainsi que les ressources multimédia sont stockés dans différents fichiers classés au sein de l'arborescence des dossiers de l'application (figure 3.1.6). Les ressources multimédia générales sont localisées dans les dossiers 'images' et 'sounds' alors qu'images et vidéos des tutoriaux sont situés dans les sous-dossiers correspondants du dossier 'tutorials'. Les frameworks, bibliothèques et autres dépendances se trouvent dans les sous-dossiers spécifiques du dossier 'librairies'.

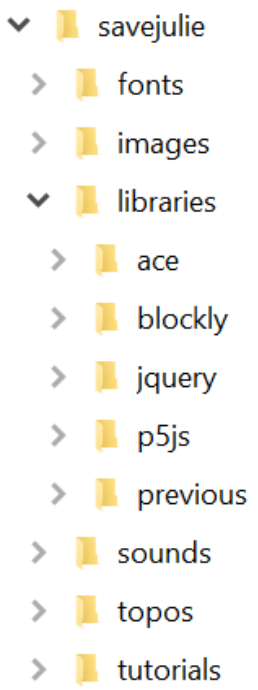


Figure 3.1.6 : arborescence des dossiers de l'application *Sauvez Julie*

Enfin, les codes sources à l'oeuvre dans l'exécution de l'application sont localisés dans ses principaux fichiers *Javascript* (tableau 3.1) :

Fichier	Dossier	Fonctions
savejulie.js	/	gestion du plateau de jeu (*play.html*) : génération, paramétrage, interactions utilisateurs avec l'interface HTML, exécution du code utilisateur et animation du topo
editor.js	/	édition de nouveaux exercices et topos au sein de l'éditeur correspondant (*editor.html*)
tutorial.js	/	extension du gestionnaire du plateau de jeu (*savejulie.js*) dédié à l'affichage et à la navigation dans le tutoriel d'un exercice lorsque celui-ci existe et est activé
blockly_interface.js	/librairies/blockly/	génération et insertion dans le plateau de jeu de l'espace de travail (éditeur) *Blockly*, enregistrement ou ouverture d'un exercice dans ou à partir d'un fichier *.json* local, lancement conversion du code visuel en code source
saveJulieBlocks.js	/librairies/blockly/	définitions des blocs visuels à disposition dans l'espace de travail *Blockly*
saveJulieGenerators.js	/librairies/blockly/	générateurs de code *Javascript*, *Python* et *Java* produisant le code source correspondant à chaque blocs constitutif du programme composé par l'utilisateur dans l'espace de travail *Blockly*
pseudo_generator.js	/librairies/blockly/	générateur de pseudo-code à partir des blocs du programme composé dans l'espace de travail Block

Tableau 3.1 : arborescence des dossiers de l'application *Sauvez Julie*

Les fonctions et instructions de ces codes sources constituent le socle d'exécution de l'interface de programmation et de l'animation du plateau de jeu.

3.2 Interface de programmation

L'interface de programmation de l'application est duale, constitué à la fois d'un éditeur de code visuel ainsi que d'un éditeur de code littéral, permettant tous deux de composer un programme générant l'itinéraire à suivre par le guide. Le code visuel peut être automatiquement traduit en code littéral dans différents langages, y compris du pseudocode. Dans sa version actuelle, seul le code écrit en Javascript peut ensuite être exécuté.

L'éditeur de code littéral est une application *Javascript* (AceEditor) dont l'interface prend la forme d'un élément *HTML* inséré dans la page *play.html*. L'application est localisée dans le sous-dossier 'ace' du dossier 'libraries' de *Sauvez Julie*. L'élément est imbriqué dans un élément de la page. Il est inséré dans cet élément par le biais de la fonction *.edit*() de l'objet *ace* instancié lors de l'importation de la bibliothèque de l'éditeur (*ace.js*). Cette fonction, recevant comme argument l'identifiant de l'élément dans lequel insérer l'éditeur ('codeEditor'), retourne un objet placé dans la variable *codeEditor* constituant l'API permettant à l'application de manipuler cet éditeur (code 3.2.1).

```
850    codeEditor = ace.edit("codeEditor");
```

Code 3.2.1 : insertion de l'éditeur *Ace* dans la page web

L'éditeur de code visuel est quant à lui également une application *Javascript* tierce (*Blockly*) intégrée dans *Sauvez Julie* et adaptée à cette dernière. L'éditeur visuel de l'application est constitué d'une interface visuelle composée d'un espace de travail (workspace) disposant, dans sa partie latérale gauche, d'une boîte à outils extensible (toolbox). La boîte à outils contient des instructions sous forme de blocs à assembler dans l'espace de travail (figure 3.2.1).

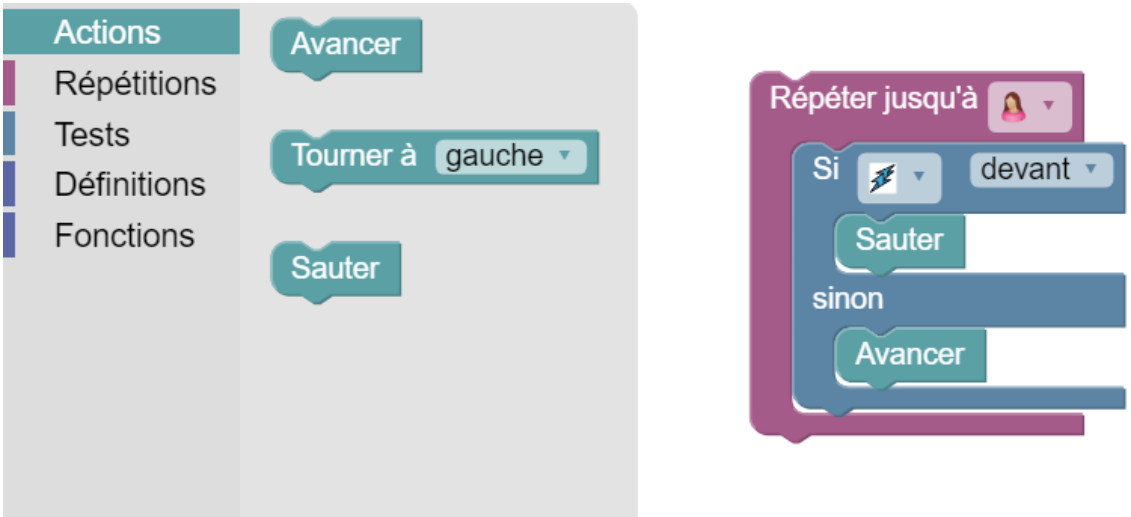


Figure 3.2.1 : interface *Blocky* - boîte à outils et espace de travail

a) éditeur visuel

L'éditeur visuel s'intègre également dans la page web au sein d'un élément HTML dont l'identifiant ('blockly-editor') est transmis à la fonction de classe Blockly *.inject*() insérant l'éditeur dans l'arbre DOM de la page web et générant un objet de classe *WorkspaceSvg*. Ce dernier, permettant à l'application de manipuler l'espace de travail, est placé dans la variable *editorWorkspace*.

La fonction *initBlocklyWorkspace*() procède à la création et à l'intégration de l'espace de travail dans la page web (code 3.2.2). Cette fonction est localisée dans le fichier 'blockly_interface.js' contenant des fonctions permettant de gérer l'éditeur visuel de l'application *Sauvez Julie* et est appelée au démarrage de l'application par la fonction *setup*() de *Processing*. Le fichier est localisé dans le dossier 'blockly' du dossier 'libraries', contenant les fichiers de l'application *Blockly*.

```
4  function initBlocklyWorkspace(){
5
6    options['toolbox'] = document.getElementById("toolbox");
7
8    editorWorkspace = Blockly.inject('blockly-editor', options);
9
10 }
```

Code 3.2.2 : insertion de l'espace de travail *Blockly* dans la page web

Un dictionnaire, situé dans la variable *options[ ]*, est également transmis à la fonction *.inject*() lors de la demande de création de l'espace de travail. Celui-ci contient les options de configuration de l'espace de travail. Sa clé 'toolbox' est une référence à un élément *XML* définissant les éléments (blocs visuels) à afficher dans la boîte à outils ainsi que leurs catégories (code 3.2.3). Cet élément *XML*, localisé dans le corps de la page web *play.html*, a comme identifiant 'toolbox'.

```
146   <xml id="toolbox" style="display: none">
147
148     <category colour="184" name="Actions">
149
150         <block type="move"></block>
151         <block type="turn"></block>
152         <block type="jump"></block>
153
154     </category>
155
156     <category colour="322" name="Répétitions">
157
158        <block type="repeat_n_times"></block>
159        <block type="repeat_until_something"></block>
160
161     </category>
162
163   </xml>
```

Code 3.2.3 : élément *XML* 'toolbox' définissant la boîte à outils (extrait)

Blockly fournit une palette de blocs standards représentant les principaux éléments fondamentaux d'un langage de programmation (instructions, variables, structures de contrôle, expression, etc). Il est cependant également possible de définir de nouveaux blocs visuels. *Sauvez Julie* dispose ainsi de de ses propres blocs. Un bloc peut être défini comme objet *JSON* ou littéralement en *Javascript*. C'est sous cette forme qu'ils sont définis dans *Sauvez Julie* au sein du fichier *saveJulieBlocks.js* localisé dans le dossier 'blockly'.

Le code d'un bloc peut être écrit manuellement ou généré automatiquement à l'aide d'un outil en ligne de 'fabrique de blocs' (*Block Factory*[2]). Celui-ci permet de modéliser visuellement un bloc (figure 3.2.2) et d'en générer son code *JSON* ou *Javascript.*

[2] https://blockly-demo.appspot.com/static/demos/blockfactory

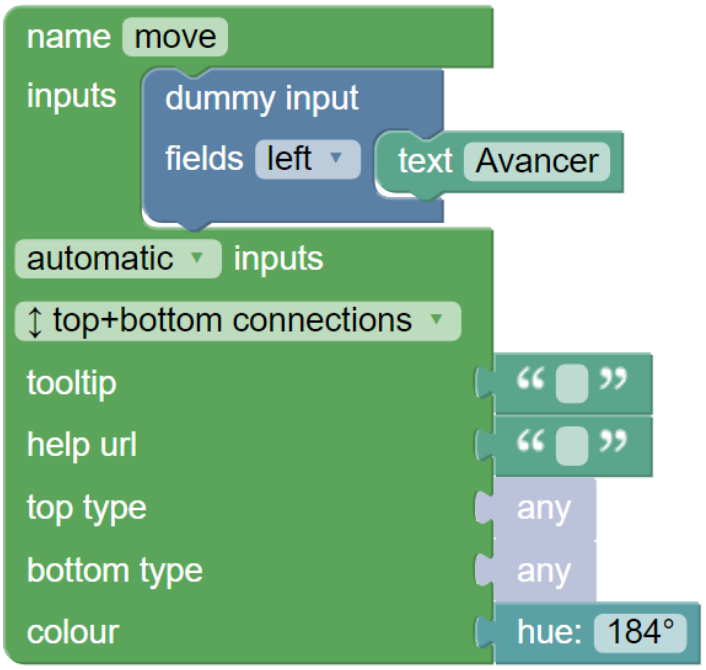


Figure 3.2.2 : modélisation du bloc 'Avancer' à l'aide de l'outil *Block Factory*

L'outil de création de nouveaux blocs génère automatiquement le code *Javascript* définissant le bloc modélisé. Celui-ci prend la forme d'un objet ajouté à la liste des blocs disponibles dans laquelle il est indexé par son nom (code 3.2.4) :

```
142 Blockly.Blocks['move'] = {
143
144   init: function() {
145
146     this.appendDummyInput()
147         .appendField("Avancer");
148     this.setPreviousStatement(true, null);
149     this.setNextStatement(true, null);
150     this.setColour(184);
151     this.setTooltip("");
152     this.setHelpUrl("");
153
154   }
155
156 };
```

Code 3.2.4 : définition *Javascript* du bloc 'Avancer'

Une fois instancié au lancement de l'application, le nouveau bloc apparaît dans la boîte à outils, au sein de la catégorie dans laquelle il a été placé, se matérialisant sous sa forme visuelle (figure 3.2.3) :

Figure 3.2.3 : bloc visuel généré par Blockly à partir de sa définition *Javascript* (code 3.2.4)

Un bloc représente une instruction d'un langage de programmation. Il peut avoir un prédécesseur et un successeur. Lorsque tel est le cas, ses parties supérieure et inférieure comprennent un connecteur. Un bloc sans connecteur inférieur est un bloc terminal. Comme toute instruction ou fonction, un bloc peut avoir des valeurs d'entrée et une valeur de sortie. Il existe deux catégories d'entrées : internes et externes.

Une *entrée externe* (dite aussi 'auxiliaire', pour 'dummy input') est une valeur saisie ou sélectionnée par l'utilisateur du bloc. Une entrée externe est composée d'un ou de plusieurs d'éléments pouvant prendre la forme d'un champ de saisie, d'une liste déroulante, d'une case à cocher ou d'autres éléments conventionnels. Un élément d'entrée est habituellement précédé d'une étiquette ('label') indiquant à l'utilisateur la nature de la valeur attendue. Les éléments d'entrées externes n'ont pas de connecteur.

A relever que (oubli des développeurs de Blockly ou volonté de sobriété conceptuelle ?), si tout bloc a un nom (valeur de son index dans la liste de blocs), celui-ci n'est pas affiché sur son bloc. Un bloc ne dispose en propre, à sa racine, d'aucun élément permettant d'y afficher du texte, à l'exception des étiquettes pouvant être placées dans une entrée externe. Afin d'afficher le nom du bloc, il est donc nécessaire de lui ajouter une première entrée externe ne contenant qu'une étiquette et aucun élément d'entrée, à l'image de ce que fait la fonction *.appendField*() du code du bloc 'Avancer' (code 3.2.4, ligne 147).

Le bloc 'Avancer' ne dispose pas d'éléments d'entrées, contrairement au bloc 'Si <objet> <direction> … sinon …' (figure 3.2.4). On trouve dans ce dernier deux éléments d'entrées externes, à savoir deux listes déroulantes, l'une permettant de sélectionner un type de terrain (crevasse paroi, piste, glacier), l'autre une orientation (devant, à gauche, à droite). Comme dans les éléments d'entrées conventionnels des interfaces graphiques des applications, ces valeurs (par exemple 'forward', 'left', 'right' pour la liste de sélection de l'orientation) ne sont pas visibles de l'utilisateur qui n'en voit que les noms correspondants. C'est cependant la valeur sélectionnée qui est retournée par l'élément d'entrée.

Si dans un champ l'utilisateur peut saisir la valeur qu'il souhaite (pour autant que la nature de l'information saisie corresponde au type de champ), les autres éléments d'entrée lui permettent de sélectionner une valeur parmi des valeurs prédéfinies.

On constate aussi bien dans le modèle du bloc (figure 3.2.4) que dans sa définition (code 3.2.5) que chaque élément d'entrée possède un nom, qui permettra par la suite d'accéder à sa valeur. Ainsi, la liste déroulante des types de terrain a comme nom 'goalField' et celle des orientations 'orientation'.

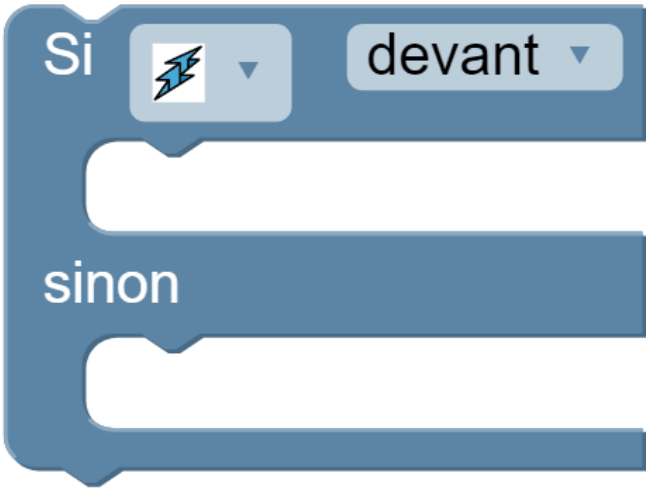


Figure 3.2.4 : bloc visuel généré par *Blockly* à partir de sa définition *Javascript* (code 3.2.5)

Les *entrées internes* sont des données issues des blocs visuels. Il peut s'agir d'une *valeur* ('value input') générée par un bloc constituant une fonction ou une expression (dans cette version, notre application ne fait pas usage de telles entrées internes de valeurs, mais cela pourrait être le cas si on lui ajoutait des blocs visuels représentant des variables et des fonctions).

Une entrée interne peut également être constituée d'un *bloc d'instructions* imbriqué ('statement input') dans le bloc défini. Tel est le cas du bloc 'Si <objet> <direction> … sinon …' qui dispose de deux entrées internes de blocs d'instructions imbriqués, l'un sous l'étiquette 'Si' et l'autre sous l'étiquette 'sinon'. Une entrée interne de bloc d'instructions possède également un nom permettant par la suite de récupérer le bloc imbriqué et son code. Ainsi, la première entrée interne du bloc <objet> <direction> … sinon …' située juste en dessous de l'étiquette 'Si' a comme nom 'statements_if' (figure 3.2.5 et code 3.2.5, ligne 82) et est constituée du bloc d'instructions à exécuter si la condition posée à l'aide des valeurs des éléments de l'entrée externe est vraie. Il en va de même avec la valeur d'entrée interne 'statements_else' pour le bloc d'instruction alternatif à exécuter.

Une fois tous les blocs définis, le joueur dispose des éléments dont il a besoin afin de composer son algorithme prenant la forme d'un code visuel.

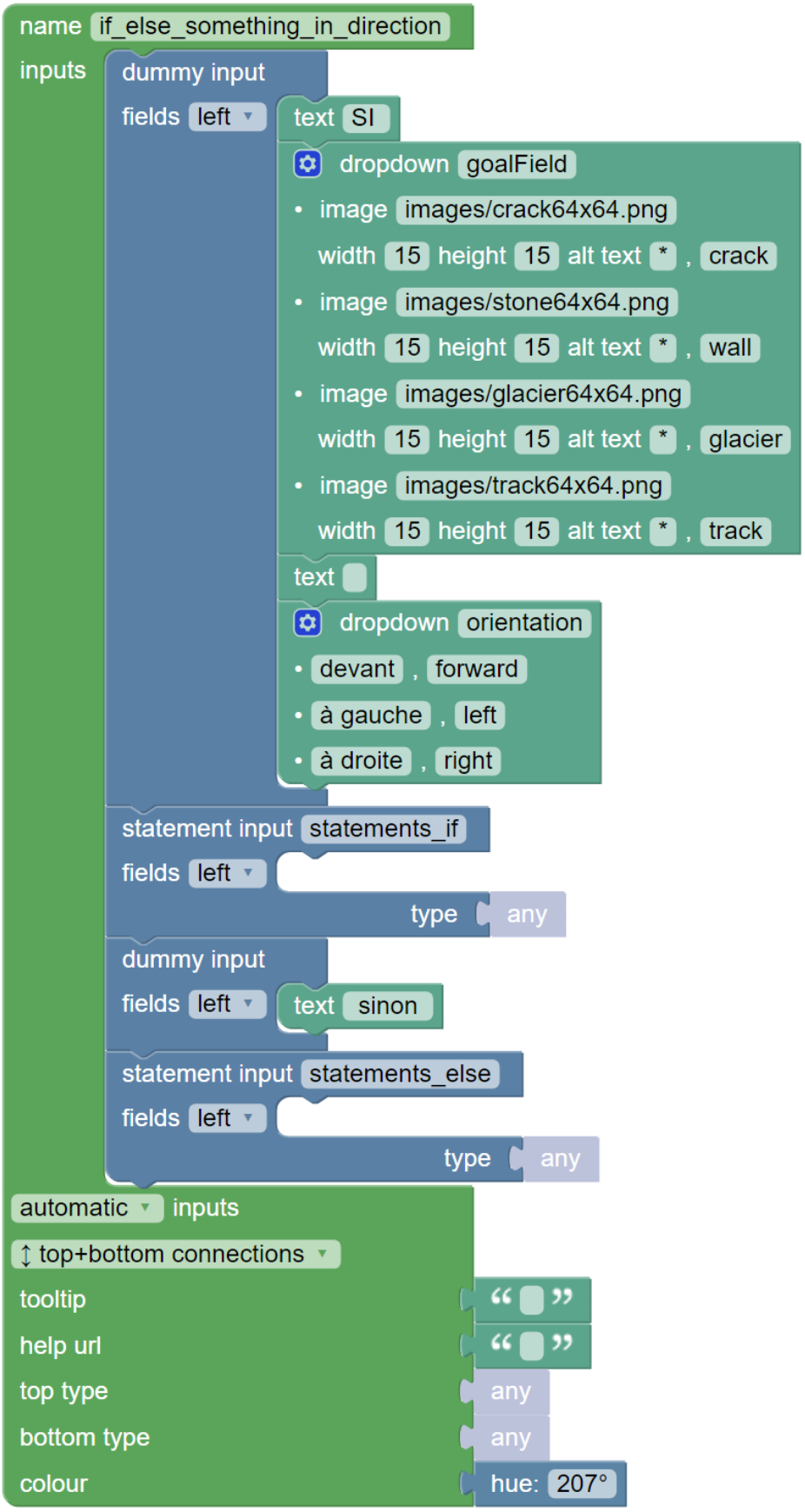


Figure 3.2.5 : modélisation du bloc 'Si <objet> <direction> … sinon …'

```
Blockly.Blocks['if_else_something_in_direction'] = {
  init: function() {
    this.appendDummyInput()
        .appendField("Si ")
        .appendField(new Blockly.FieldDropdown(

         [
            [{"src":"images/crack_white_128x128.png",
              "width":15,"height":15,"alt":"*"},"crack"],

            [{"src":"images/stone128x128.png",
              "width":15,"height":15,"alt":"*"},"wall"],

            [{"src":"images/glacier32x32.png","width":15,
              "height":15,"alt":"*"},"glacier"],

            [{"src":"images/track32x32.png",
              "width":15,"height":15,"alt":"*"},"track"]
         ]), "goalField")

        .appendField(" ")
        .appendField(new Blockly.FieldDropdown(

         [["devant","forward"], ["à gauche","left"],
          ["à droite","right"]]), "orientation");

    this.appendStatementInput("statements_if")
        .setCheck(null);
    this.appendDummyInput()
        .appendField("sinon");
    this.appendStatementInput("statements_else")
        .setCheck(null);
    this.setPreviousStatement(true, null);
    this.setNextStatement(true, null);
    this.setColour(207);
    this.setTooltip("");
    this.setHelpUrl("");
  }
};
```

Code 3.2.5 : définition Javascript du bloc 'Si <objet> <direction> … sinon …'

b) conversion du code visuel en code littéral

Une fois le programme composé visuellement, l'ensemble de ses blocs doit être traduit dans un langage de programmation pouvant être exécuté par la machine. Par défaut, notre application fait cette traduction en *Javascript* mais il est possible de lui demander de générer le code source à exécuter dans d'autres langages, à savoir ici *Python* et *Java*. Cependant, s'agissant d'une application web exécutée exclusivement dans un navigateur, *Sauvez Julie* exécute uniquement du code écrit en *Javascript*. Une version ultérieure pourrait cependant être à même d'exécuter du code rédigé dans d'autres langages, en faisant appel à des bibliothèques de transcodage, ou, mieux, à des compilateurs en *WebAssembly*.

Avant de pouvoir exécuter son programme visuel, le joueur doit donc dans un premier temps en demander la traduction en code source. C'est ce qu'il fait en cliquant sur le bouton 'Coder' (crayon) en dessous de l'éditeur de code *Ace*. Cela appelle la fonction *generateCode*() du script *blockly_interface.js*.

```
115    switch(language){
116
117     case "javascript":
118
119        code[worldNb] = Blockly
120                        .JavaScript
121                        .workspaceToCode(editorWorkspace);
122        break;
123
124     case "python":
125
126        code[worldNb] = Blockly
127                        .Python
128                        .workspaceToCode(editorWorkspace);
129        break;
130
131     case "java":
132
133        code[worldNb] = Blockly
134                        .Java
135                        .workspaceToCode(editorWorkspace);
136        break;
137
138     case "pseudo" :
139
140        code[worldNb] = Blockly
141                        .Pseudo
142                        .workspaceToCode(editorWorkspace);
143
144    }
```

Code 3.2.6 : fonction *generateCode*() - appel à la fonction .*workspaceToCode*()

La fonction *generateCode*() appelle alors la fonction .*workspaceToCode*() du générateur de code du langage sélectionné, lui-même membre statique de la classe *Blockly* (code 3.2.6). A noter en passant que du pseudo-code peut également être généré à partir des blocs visuels. La fonction .*workspaceToCode*() reçoit comme argument une référence à l'espace de travail dans lequel se trouvent les blocs à traduire.

La fonction .*workspaceToCode*() retourne le code source du code visuel (blocs) retranscrit dans le langage de programmation courant. Le code retourné est placé dans la liste *code*[ ] avec comme valeur d'index le numéro du topo courant pour le parcours duquel l'utilisateur a conçu l'algorithme se trouvant dans l'espace de travail.

Cette fonction du générateur de code n'est cependant pas à même de traduire littéralement le code visuel dans le langage correspondant à la classe à laquelle elle appartient. Encore faut-il instancier ce générateur de code ainsi qu'une fonction de traduction pour chaque bloc visuel disponible. *Blockly* est fourni avec des blocs de base correspondant aux éléments fondamentaux que l'on trouve dans la plupart des langages de programmation. Ces blocs peuvent être personnalisés tout comme il est possible d'ajouter des blocs additionnels, comme nous l'avons fait dans *Sauvez Julie*.

Blockly fournit des générateurs de code pour cinq langages de programmation : *JavaScript*, *Python*, *PHP*, *Dart*, et *Lua*. Il est cependant possible d'ajouter dans un projet un générateur de code pour un langage additionnel. Le constructeur de la sous-classe *Generator*() de la classe *Block* instancie un nouveau générateur de code que l'on peut ajouter comme objet membre statique de la classe *Blockly*. Ainsi la seconde instruction du script *pseudo_generator.js* (code 3.2.7, ligne 3) instancie-t-elle un générateur de pseudo-code. Un générateur de code a habituellement comme nom celui correspondant au langage de programmation dans lequel il effectue le codage.

```
3 Blockly.Pseudo = new Blockly.Generator('Pseudo');
```

Code 3.2.7 : constructeur d'un générateur de code - générateur de pseudo-code

Des instructions correspondantes se trouvent dans les scripts contenant les générateurs de code des langages de programmation supportés par *Sauvez Julie*. Ceux-ci contiennent également les fonctions de génération de code de chaque bloc standard. Notre application utilisant des blocs lui étant spécifiques définis dans le script *saveJulieBlocks.js*, encore faut-il ajouter à chaque générateur de code des langages mis à disposition par l'application des fonctions générant le code spécifique à chacun de ces blocs. Ces fonctions sont localisées dans le script *saveJulieBlocks.js* pour *Javascript*, *Python* et *Java* ainsi que dans *saveJulieBlocks.js* pour le pseudo-code.

Un générateur de code est constitué d'une liste des fonctions de codage des blocs visuels. Chaque fonction est indexée avec le nom du bloc dont elle assure le codage. Une fonction de codage retourne le code du bloc visuel auquel elle est attachée. Le codage du bloc 'Avancer' ('move') (figure 3.2.3) génèrera par exemple un code *Javascript* composé d'une

seule instruction, un appel à la fonction *move*() demandant au guide d'avancer d'une case sur la glacier (code 3.2.8)

```
1 Blockly.JavaScript['move'] = function(block) {
2   var code = 'move();\n';
3   return code;
4 };
```

Code 3.2.8 : fonction de codage *Javascript* du bloc 'Avancer'

On peut observer que la fonction reçoit comme argument le bloc à coder. S'il est appelé à être visualisé ou édité, la fonction doit également se soucier du formatage du code. Dans l'exemple qui précède, on constate que l'instruction *move*() est suivi d'un caractère de contrôle provoquant un retour à la ligne (\n). Tel est également le cas lorsque le formatage du code source fait partie formellement de sa structuration, comme c'est le cas avec *Python* (indentation obligatoire des blocs d'instructions et des imbrications).

Les instructions élémentaires pouvant être données au guide (avancer, tourner, sauter) sont définies dans le script *savejulie.js* : *move*(), *turnLeft*(), *turnRight*(), *jump*(). Des structures de contrôles spécifiques permettent de les encadrer. Celles-ci se présentent visuellement sous la forme de blocs dédiés tels que les blocs 'Répéter jusqu'à <objet>' ou 'Si <objet> <direction> … sinon …'. Le code généré pour de tels blocs ne se limite pas à une seule instruction, comme dans le cas du bloc 'Avancer'. Il fait également appel aux structures de contrôle de base du langage de programmation correspondant. De plus, pareils blocs structurés contiennent un ou plusieurs blocs imbriqués, dont il s'agit également d'imbriquer le ou les ou les codes correspondant au sein du code de ces blocs.

Illustrons ceci avec la définition de la fonction de codage Javascript d'un bloc 'Si <crevasse> <devant> … sinon …' (figure 3.2.6 et code 3.2.9abc).

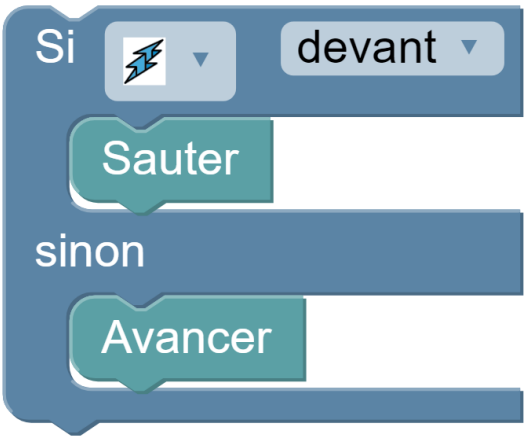


Figure 3.2.6 : bloc 'Si <crevasse> <devant> … sinon …'

Dans un premier temps, la fonction de codage va accéder à la première entrée interne du bloc, à savoir au premier bloc d'instructions imbriqué situé en dessous de l'entrée externe 'Si'. Dans la définition du bloc 'Si <objet> <direction> … sinon …' le nom 'statements-if' a été attribué à cette entrée. (code 3.2.5, ligne 82). La fonction *.statementToCode*() du générateur de code retourne le code d'un bloc d'instructions imbriqués dans un bloc donné. C'est ce que fait son premier appel, retournant le code du bloc d'instructions imbriqué situé dans l'entrée interne ayant comme l'identifiant 'statements_if' (code 3.2.9a, lignes 227 à 229). Le code retourné est placé dans la variable locale *statements_if*.

La fonction de codage en fait de même avec le code du second bloc imbriqué ayant comme identifiant 'statements_else' (code 3.2.9a, lignes 231 à 233). Elle récupère également les deux valeurs externes du bloc par le biais de la fonction *.getFieldValue*(), à savoir celle ayant comme identifiant 'goalField' – dont la valeur représente le type d'objet dont il faut vérifier la proximité avec le guide ('crack', 'wall', 'glacier', 'track')) – ainsi que celle identifié comme 'orientation' – sa valeur représentant l'orientation de la proximité ('forward', 'left', 'right'). Les deux valeurs sont placées dans les variables locales *goal* et *orientation*. (code 3.2.9a, lignes 235 et 236).

```
223 Blockly.JavaScript['if_else_something_in_direction']
224
225   = function(block) {
226
227   var statements_if = Blockly
228                       .JavaScript
229                       .statementToCode(block, 'statements_if');
230
231   var statements_else = Blockly
232                         .JavaScript
233                         .statementToCode(block, 'statements_else');
234
235   var goal = block.getFieldValue('goalField');
236   var orientation = block.getFieldValue('orientation');
237   var code = '\n';
238
```

Code 3.2.9a : fonction de codage *Javascript* du bloc 'Si <objet> <direction> … sinon …' (i)

La variable locale *code* est également initiée avec un retour à la ligne. Elle est destinée à recevoir le code du bloc généré par les instructions la suivant. Celui-ci sera fonction des valeurs d'entrée du bloc. L'instruction *if()* injectée dans le code testera la véracité de la valeur retournée par différentes fonctions selon le type d'objet dont on souhaite vérifier la proximité : *isCrack*(), *isWall*(), *isGlacier*(), *isTrack*(). La fonction retenue recevra en argument l'orientation de la proximité à tester, à savoir l'objet dont on souhaite tester la proximité devait se trouver devant, sur la gauche ou sur la droite du guide (code 3.2.9b).

```
239   switch(goal){
240
241     case "crack" :
242
243       code += 'if(isCrack("' + orientation +'")){\n';
244       break;
245
246     case "wall" :
247
248       code += 'if(isWall("' + orientation +'")){\n';
249       break;
250
251     case "glacier" :
252
253       code += 'if(isGlacier("' + orientation +'")){\n';
254       break;
255
256     case "track" :
257
258       code += 'if(isTrack("' + orientation +'")){\n';
259       break;
260
261   }
```

Code 3.2.9b : fonction de codage *Javascript* du bloc 'Si <objet> <direction> … sinon …' (ii)

La suite du code à générer est indépendante des valeurs des entrées externes. Elle contient dans tous les cas le code du bloc d'instructions imbriqué à exécuter si le test s'est révélé positif, code déjà généré et placé dans la la variable *statements_if*. Celui-ci est ajouté au contenu de la variable *code*, en étant intégré à l'instruction *if*() comme bloc d'instructions à exécuter en cas de test positif (code 3.2.9c, lignes 263 à 266). La fonction de codage en fait ensuite de même pour le code du bloc d'instructions alternatif à exécuter en cas de test négatif (lignes 266 à 270).

```
263   code += '\n';
264   code += statements_if;
265   code += '\n';
266   code += '} else {\n';
267   code += '\n';
268   code += statements_else;
269   code += '\n';
270   code += '}\n';
271   code += '\n';
272
273   return code;
274
275   };
```

Code 3.2.9c : fonction de codage *Javascript* du bloc 'Si <objet> <direction> … sinon …' (iii)

L'intégralité du code généré est alors retourné par la fonction (figure 3.2.7)

```
if(isCrack("forward")){

  jump();

} else {

  move();

}
```

Figure 3.2.7 : code *Javascript* généré pour le bloc 'Si <crevasse> <devant> … sinon …'

La fonction .*workspaceToCode*() d'un générateur de code (code 3.2.6) lit séquentiellement et par récursivité chacun des blocs visuels d'un programme composé dans l'espace de travail, fait appel à leurs fonctions de codage respective et concatène les codes retournés afin de produire le code complet du programme. Celui-ci est ensuite introduit dans l'éditeur de code. Il pourra être lu et édité par le joueur ainsi qu'exécuté par le moteur d'animation (figure 3.2.8).

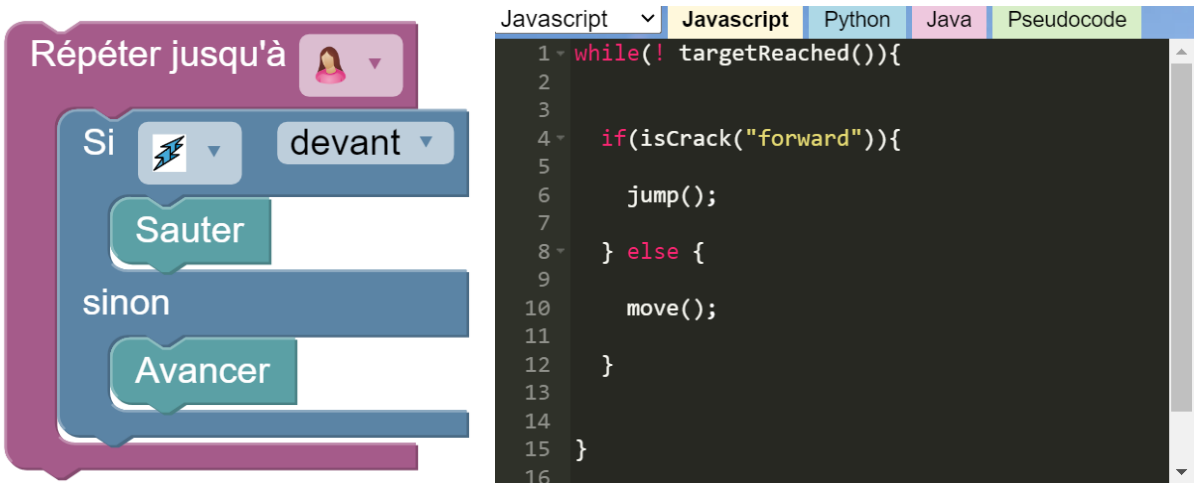


Figure 3.2.8 : génération du code d'un programme par la fonction .*workspaceToCode*()

3.3 Moteur d'animation

a) représentation graphique d'un monde

L'exécution du code produit par le joueur donne lieu à une animation. Le guide exécute les opérations qu'on lui demande d'effectuer afin de suivre l'itinéraire déterminé par le joueur. Cela se traduit par une séquence de mouvements et déplacements sur le plateau de jeu représentant la surface du glacier, sous la forme d'une grille de cellules.

Chaque aventure est composée de plusieurs scènes. A chaque scène correspond une topographie (topo) spécifique d'un glacier, avec ses zones de glace vive, ses crevasses et ses zones recouvertes de neige. Dans le jargon du code de l'application, chaque scène constitue un *monde* (world). L'ensemble des mondes sont réunis dans un *univers* (universe).

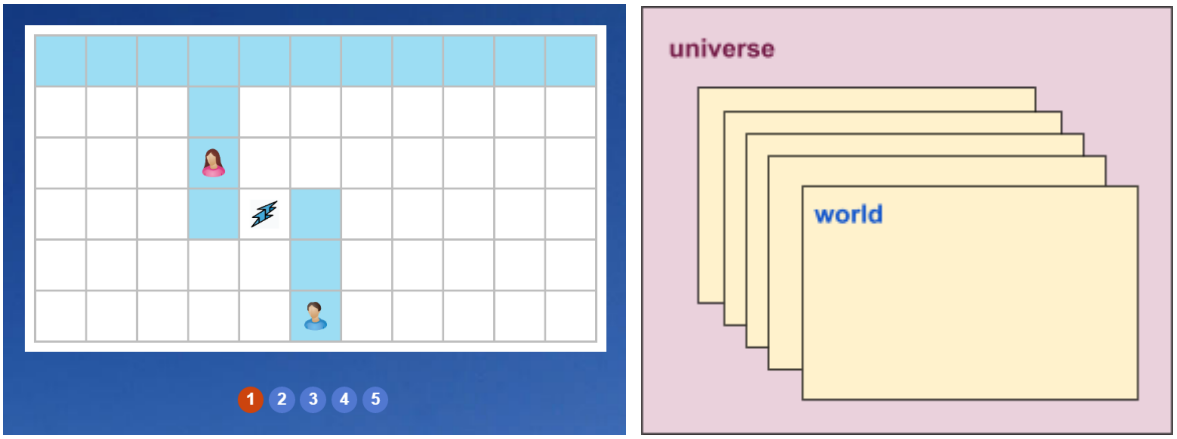


Figure 3.3.1 : univers, mondes et topos

Un univers prend la forme d'un objet stocké dans un fichier ***.json***, manipulé à partir de la variable le contenant et chargé à partir de ce fichier. L'attribut *.info* contient un objet de classe *Universe* dans lequel on trouve des métadonnées au sujet de l'univers spécifique (nom, auteur, tutoriel attaché à cet univers). L'ensemble des mondes d'un univers est constitué d'un tableau localisé dans l'attribut *.data*. Les attributs d'un monde spécifient son nom (*.title*), sa topographie (*.world*), l'orientation initiale (nord, est, sud, ouest) du guide (*playerOrientation*) et l'état de réussite d'une scène (*.succeed*). Les blocs décrivant l'itinéraire que le guide est appelé à suivre sont mémorisés en format ***.xml*** dans l'attribut *.blocks* alors que le code *Javascript* correspondant l'est dans l'attribut *.javascript*. La figure 3.3.2 ci-dessous récapitule la structure des données constituant un univers :

Figure 3.3.2 : structure des données d'un univers

La figure 3.3.3 illustre les valeurs des différents attributs d'un univers d'apprentissage de la notion de séquences d'instructions (extrait du fichier correspondant) :

```
{
  "info": {
    "name": "Exercice 1 - séquences",
    "author": "Collège St-Michel",
    "currentItem": 0,
    "tutorial": "sequences",
    "activeTutorial": true,
    "loadedTutorial": false
  },
```

```
"data": [
  {
    "title": "Séquence d'instructions I",
    "succeed": false,
    "playerOrientation": 1,
    "world": [["o", "o", "o", "o", "o", "o", "o"],
              ["o", "o", "o", "o", "o", "o", "o"],
              ["p", "t", "x", "x", "x", "x", "x"],
              ["o", "o", "o", "o", "o", "o", "o"],
              ["o", "o", "o", "o", "o", "o", "o"]
             ],
    "blocks": "<xml xmlns=\"http://www.w3.org/1999/xhtml\"></xml>",
    "javascript": ""},
  {
    "title": "Séquences d'instructions XVIII",
    "succeed": false,
    "playerOrientation": 0,
    "world": [["p", "c", "x", "c", "x", "x", "c", "x", "c", "x"],
              ["o", "x", "o", "x", "o", "x", "o", "x", "o", "c"],
              ["x", "o", "o", "o", "c", "o", "x", "o", "o", "x"],
              ["c", "o", "x", "c", "x", "o", "t", "o", "o", "x"],
              ["x", "o", "x", "o", "o", "o", "o", "x", "o", "o"],
              ["x", "o", "o", "x", "x", "x", "x", "x", "o", "x"],
              ["o", "c", "o", "o", "o", "c", "o", "o", "c", "o"],
              ["x", "o", "x", "c", "x", "o", "x", "x", "o", "x"]
             ],
    "blocks": "<xml xmlns=\"http://www.w3.org/1999/xhtml\"></xml>",
    "javascript": ""}
]
}
```


Figure 3.3.3 : extrait des attributs d'un univers

La topographie (topo) d'un monde (attribut *.world*) est constituée d'un tableau de caractères à deux dimensions. Chaque caractère représente symboliquement la nature de la cellule correspondante (figure 3.3.4) :

Symbole	Signification	Icone
'x'	zone du glacier sûre (sans neige)	
'o'	zone du glacier recouverte de neige	
'c'	crevasse ouverte	

Symbole	Signification	Icone
's'	rocher	
't'	Julie (touriste)	
'p'	Marc (guide)	
'f'	Jalon (flag) (équivalent à 't')	
'-'	Hors carte	

Figure 3.3.4 : symboles des cellules d'un topo

La fonction ***drawWorld()*** de la bibliothèque *savejuilie.js* projette sur le canevas le topo du monde courant :

```
[["o", "t", "x", "o", "o", "o", "o"],
 ["o", "o", "o", "o", "o", "o", "o"],
 ["o", "o", "x", "c", "x", "x", "o"],
 ["o", "o", "o", "o", "o", "x", "o"],
 ["o", "o", "o", "o", "o", "p", "o"]
]
```

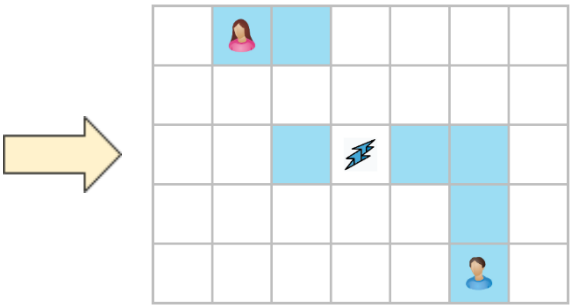


Figure 3.3.5 : représentation graphique d'un topo généré par la fonction *drawWorld*()

L'animation graphique consiste à projeter sur le canevas, à intervalles réguliers, un topo dont le contenu varie. Dans le cas de notre jeu, c'est la position du guide qui va changer séquentiellement, celui-ci passant d'une case à une autre en suivant l'itinéraire que le joueur lui demande de suivre. Afin de procéder à cette animation, il est donc nécessaire dans un premier temps de générer cet itinéraire à partir du code composé par le joueur, puis de le faire suivre par le guide.

b) animation graphique d'un monde

La **classe *Player*** permet d'instancier un objet représentant le joueur, endossant le rôle de Marc le guide. Cet objet correspond à un pion sur le plateau de jeu. Ses attributs précisent l'état du pion, à l'image de ses coordonnées sur le glacier, l'itinéraire à suivre ou le chemin déjà parcouru. Ses méthodes sont utilisées afin de donner des instructions au guide quant à

l'itinéraire à suivre. Le guide prend note de l'intégralité de cet *itinéraire* en enregistrant la suite d'actions élémentaires (avancer, sauter, tourner) qu'il sera appelé à effectuer séquentiellement dans un second temps lorsqu'il se mettra en route. Les instructions de déplacements (avancer, tourner, sauteur) peuvent ainsi être données au guide dans deux modes distincts que sont le mode 'enregistrement' (*recording*) et le mode 'animation' (*playing*).

1. *mode 'enregistrement'*

Il est actif lorsque l'attribut 'recording' de l'objet de classe Player est à *vrai*. Dans ce mode, les instructions sont données au guide à l'aide de ses méthodes *.move*(), *.turnLeft*(), *turnRight*() et *.jump*(). A l'appel de chacune de ces méthodes en mode 'enregistrement), un code d'instruction (bytecode, figure 3.3.6) correspondant est ajouté à une liste d'instructions (attribut 'statements') :

méthode	**code d'instructions**
.move()	**'m'**
.jump()	**'j'**
.turnRight()	**'r'**
.turnLeft()	**'l'**

Figure 3.3.6 : codes d'instructions intermédiaires (bytecode) d'un itinéraire

Les méthodes *.turnLeft*() et *.turnRight*() en font de même. Tourner sur soi-même n'engendrant pas un déplacement critique (potentiellement dangereux), contrairement à *.move*() ou *.jump*(), le traitement de ces mouvements n'est pas réalisé par la méthode *.go*(). Cette dernière s'occupe de traiter les commandes de déplacement en vérifiant si celles-ci mènent ou non le guide dans une zone à risque.

Afin d'éviter au joueur d'assimiler des notions de programmation objet, la bibliothèque *savejulie.js* met à disposition des fonctions d'interface correspondantes permettant d'appeler les méthodes de l'objet de classe Player : ***move*()**, ***jump*()**, ***turnRight*()** et ***turnLeft*()**. Ces fonctions appellent les méthodes de l'objet 'player' auquel elles sont associées, évitant ainsi au joueur de devoir écrire des instructions faisant appel à cet objet (telles que 'player.*move*()' ou 'player.*jump*()).

Lors de l'exécution des instructions du code écrit dans l'éditeur, les méthodes élémentaires de déplacement sont appelées en mode 'enregistrement' : le code source est traduit en bytecode constitué exclusivement d'instructions élémentaires insérés dans la liste d'instructions de l'attribut *.statements*. Ce code source est ainsi transformé en une séquence d'instructions élémentaires linéaire épurée de ses structures de contrôle.

La séquence de bytecode générée pourra différer, pour un même code source, d'un topo à un autre. Ainsi, appliquée à ce topo (figure 3.3.7) :

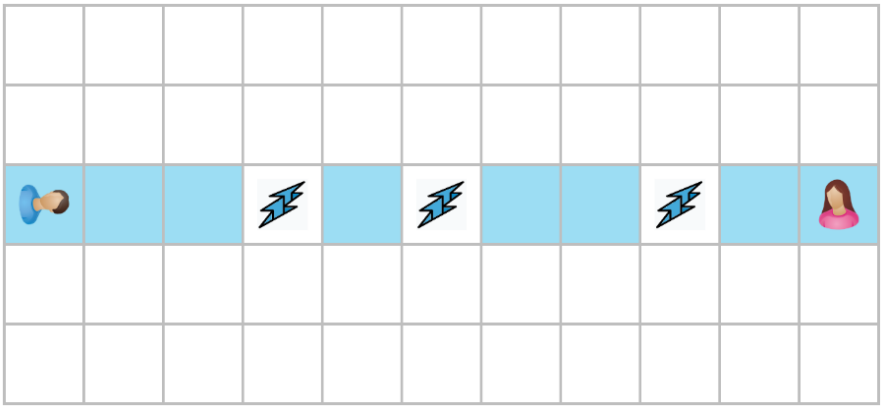


Figure 3.3.7 : topo linéaire crevassé

Le code source suivant (figure 3.3.8) :

Javascript | Javascript | Python | Java | Pseudocode

```
1  while(! targetReached()){
2
3
4    if(isCrack("forward")){
5
6      jump();
7
8    } else {
9
10     move();
11
12   }
13
```

Figure 3.3.8 : code source pour un itinéraire sur un topo linéaire crevassé

sera séquencé sous la forme du bytecode (figure 3.3.9) :

No	code d'instruction
0	'm'
1	'm'
2	'j'
3	'j'

No	code d'instruction
4	'm'
5	'j'
6	'm'

Figure 3.3.9 : bytecode pour un itinéraire sur un topo linéaire crevassé spécifique

Le bytecode est généré par les méthodes **.go**(), .**turnRight**() et .**turnLeft**(). La méthode .*go*() est appelée par les méthodes .*move()* et .*jump().* lorsqu'elles sont exécutées en mode 'enregistrement'. Elle ajoute à la liste d'instructions *statements*[ ] le bytecode correspondant à l'argument qu'elle a reçu, soit 'move' ou 'jump' (code 3.3.1).

```
912   this.go = function(motion){
913
914     if(motion === "move"){
915
916       stepNb = 1;
917       statementCode = "m";
918
919     } else if(motion === "jump") {
920
921       stepNb = 2;
922       statementCode = "j";
923
924     } else {
925
926       return;
927
928     }
929
930     if(this.recording){
931
932       this.statements.push(statementCode);
```

Code 3.3.1 : fonction .*go*() de la classe *Player* - insertion d'un déplacement

Contrairement au simple changement de direction, une avancée (move) ou un saut (jump) constituent des opérations critiques pour le guide. Un tel mouvement peut amener le guide dans une zone (cellule) dangereuse (crevasse ouverte ou couverte de neige, base d'une paroi rocheuse instable) dans laquelle il aura un accident (chute) mettant un terme à son itinéraire.

Pour chaque nouveau pas ou saut, la méthode *.go*() vérifie, en mode 'enregistrement', que la prochaine cellule à laquelle le mouvement aboutit ne soit pas une cellule à risque. Si le mouvement suivant amène le guide sur une telle cellule. l'enregistrement de l'itinéraire du guide sous la forme d'une séquence d'instructions élémentaires (bytecode) se termine ('recording' ← false) et le mode 'action' est activé ('playing' ← true) afin de basculer dans le mode d'animation.

La fonction *run*() (code 3.3.2) exécute le code source produit par le joueur et contenu dans l'éditeur. L'exécution de ce code source fait appel aux méthodes *.go*(), *.turnLeft*() et *.turnRight*() en mode enregistrement .

La variable *data* est un raccourci vers l'attribut *.data* de l'objet *universe*. La variable *worldNb* spécifie le numéro du topo actif.

```
1654  function run(){
1655
1656    data[worldNb]["javascript"] = codeEditor.getValue();
1657
1658    breakingCode = breakCode(data[worldNb]["javascript"]);
1659
1660    runScript(breakingCode);
1661
1662    started = false;
1663
1664    if(player){
1665
1666      loop();
1667
1668    }
1669
1670  }
```

Code 3.3.2 : fonction *run*() (bibliothèque 'savejulie.js')

La fonction ***breakCode()*** ajoute au début de chaque bloc d'instructions d'une structure de contrôle de répétition *while*() une structure de branchement *if*() permettant d'intercepter une potentielle boucle infinie qui ferait perdre le contrôle de l'application. Le code ajouté compte le nombre de répétitions à l'aide de la variable *playerCodeLoopCount*, réinitialisée à 0 par la fonction *breakCode*() avant l'exécution du code source protégé. Si le nombre de répétitions engendré par la structure *while*() est supérieur au nombre de répétitions tolérées par l'application et stocké dans la constante *maxPlayerCodeLoop* (par défaut : 100), l'exécution du code source est interrompue (return) et une fenêtre surgissante (popup) avertit le joueur du risque potentiel identifié (figure 3.3.10).

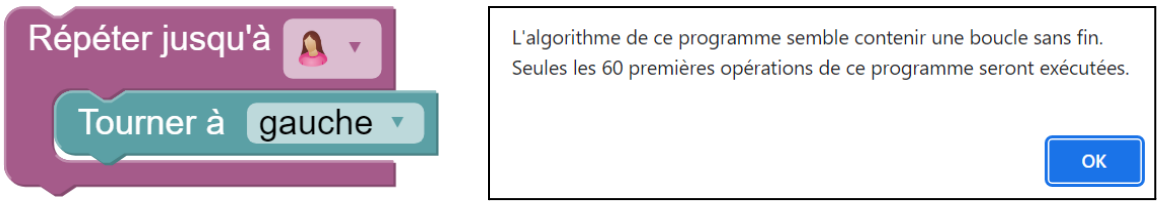


Figure 3.3.10 : interruption d'une boucle infinie au sein d'une structure *while*()

Le code source de la figure 3.3.8 se présentera ainsi, une fois étendu et protégé à l'aide de la fonction *breakCode*() (code 3.3.3) :

```
while(! targetReached()){

  if(playerCodeLoopCount++ > maxplayerCodeLoop){

    neverendingLoopDetected();
    return;

  }

  if(isCrack("forward")){

    jump();

  } else {

    move();

  }

}
```

code 3.3.3 : code source avec protection contre les boucles infinies

Afin d'exécuter le code source étendu, la fonction *run*() appelle la fonction **runScript()** (code 3.3.4). Celle-ci génère dynamiquement une nouvelle fonction *execSourceCode*() retournée par le constructeur de la classe Javascript **Function**. Cette fonction est insérée dans le contexte d'exécution courant de l'application, ayant ainsi accès à toutes ses ressources en leur état du moment (constantes, variables, fonctions).

```
1832 function runScript(code){
1833
1834   try{
1835
1836     let execSourceCode = new Function(code);
1837
1838     execSourceCode();
1839
1840   } catch(e){
1841
1842     alert(e.message);
1843
1844   }
1845
1846 }
```

Code 3.3.4 : fonction *runScript*()

Les instructions du code source sont exécutées instantanément. Cette exécution sans délai ni intervalle de temps ne peut ainsi être utilisée afin de produire une animation. Elle se contente de générer l'itinéraire à suivre, se matérialisant sous la forme d'une liste de codes d'instructions élémentaires de mouvement (séquence de bytecode). Une fois le code source exécuté et l'itinéraire généré, la fonction *run*() fait démarrer le moteur d'animation du framework *Processing* à l'aide de l'instruction *loop*() (figure 3.3.2, lignes 1664 à 1668, lançant l'exécution en boucle et à intervalle régulier de la fonction *draw*() et faisant passer l'objet *player* du mode 'enregistrement' au mode 'action'.

2. *mode 'action'*

Dans le mode 'action' (playing), le guide suit l'itinéraire décrit dans la table .*statements* sous forme d'une séquence de bytecode représentant les mouvements élémentaires du trajet à réaliser. Il bouge et se déplace physiquement sur le topo du glacier. Un moteur d'animation exécute à intervalle régulier les instructions du tableau de bytecode, constituant l'itinéraire à suivre (figure 3.3.11).

No	code
0	'm'
1	'm'
2	'j'
3	'j'
4	'm'
5	'j'
6	'm'

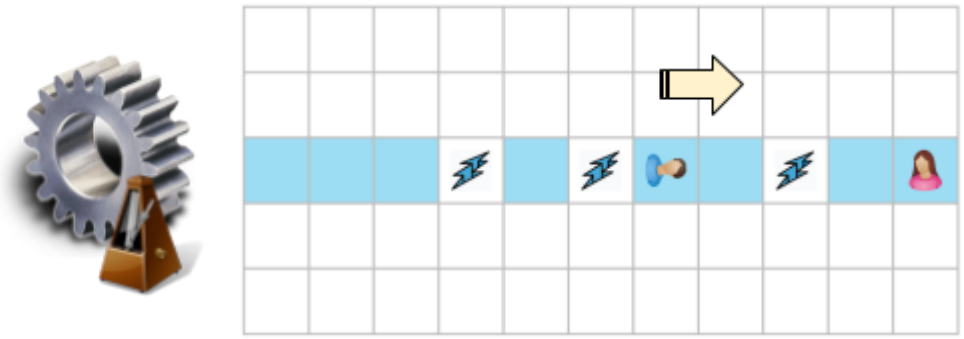


Figure 3.3.11 : liste d'instructions (itinéraire) et moteur d'animation

Dans le cadre de l'environnement d'animation *Processing* (p5js), la fonction ***draw***() (code 3.3.5) est exécutée, par défaut, 60 fois par seconde. Elle permet de redessiner à intervalle régulier le canevas sur lequel est affichée la grille de jeu. Si, à ce même intervalle de temps, la position du guide change dans le topo projeté (variable *world*) sur la grille du canevas, nous obtenons une animation.

```
872 function draw() {
873
874   if(started){
875
876     player.playNext();
877     drawWorld();
878
879   } else {
880
881     started = true;
882     frameRate(framesRate);
883
884   }
885
886 }
```

Code 3.3.5 : fonction *draw*() comme moteur d'animation

Dans le cas de notre application, la fonction *draw*() regarde si c'est la première fois qu'elle est appelée. La variable d'état *started* contient par défaut la valeur initiale *false*. Celle-ci

passe à *true* lors de ce premier appel. La vitesse de l'animation est alors fixée à l'aide de l'instruction *frameRate*() dont l'argument indique combien de fois par seconde l'image du topo sera projetée sur le canevas (par défaut : 1.5, soit un taux de rafraîchissement de l'affichage d'un peu plus d'une fois par seconde).

Cette initialisation permet par la même occasion d'instaurer un délai d'environ une seconde entre le moment où le joueur clique sur le bouton pour démarrer l'exécution du code et le premier déplacement du guide vers la prochaine cellule, la fonction *draw*() étant à nouveau appelée une seconde fois après ce laps de temps.

Par la suite, la méthode ***.playNext***() de l'objet *player* est appelée à intervalle régulier (1.5 fois par seconde) par la fonction .*draw*().

```
1504   this.playNext = function(){   // plays next motion animation sequence
1505
1506     var statement;
1507
1508     if(this.missionSuccess){
1509
1510       succeedSound.play();
1511
1512       noLoop();
1513
1514       return;
1515
1516     }
```

Code 3.3.6a : méthode .*playNext*() de la classe *Player* (i)

Si le mouvement précédent a permis au guide de rejoindre la touriste, l'attribut .*missionSuccess* voit sa valeur passer de *false* à *true* (code 3.3.6a). Le cas échéant, l'animation s'arrête. La fonction *noLoop*() met un terme à l'appel à intervalle régulier de la fonction *draw*(). Si tel n'est pas le cas, la méthode .*playNext*() s'assure que toutes les instructions de la liste de mouvements n'ont pas encore été exécutées (code 3.3.6b, ligne 1518). En ce cas, elle extrait le prochain code d'instruction à exécuter (ligne 1520) et demande à la méthode correspondante de s'en charger (lignes 1525 à 1539) en mode 'action' (playing). Dans le cas contraire (lignes 1545ss), une fois l'intégralité de l'itinéraire parcouru par le guide sans pour autant qu'il rejoigne la touriste, la mission est un échec : cet état est relevé dans l'attribut .*missionFailed* passant de false à *true*, l'animation est arrêté, le topo est réinitialisé et affiché dans sa version d'origine (appel de la fonction *initializeWorld*()), le guide étant à nouveau placé dans sa case initiale de départ (lignes 1545 à 1561).

```
1518      if(this.seqNb < this.statements.length){
1519
1520        statement = this.statements[this.seqNb];
1521
1522        this.recording = false;
1523        this.playing = true;
1524
1525        switch(statement){
1526
1527          case "m" : this.move();
1528                     break;
1529
1530          case "j" : this.jump();
1531                     break;
1532
1533          case "l" : this.turnLeft();
1534                     break;
1535
1536          case "r" : this.turnRight();
1537                     break;
1538
1539        }
1540
1541        this.seqNb++;
1542
1543        return true;
1544
1545      } else {
1546
1547        noLoop();
1548
1549        if(this.playing){
1550
1551          this.missionFailed = true;
1552          allSoundsfadeOut(2);
1553          failedMission.play();
1554          setTimeout(reinitializeWorld, 3000);
1555
1556        }
1557
1558
1559        return false;
1560
1561      }
1562
1563    }
```

Code 3.3.6b : méthode *.playNext*() de la classe *Player* (ii)

Lors de l'exécution d'un nouveau déplacement du guide, la méthode *.go*() est appelée par l'intermédiaire des méthodes *.move*() ou *jump*(). La méthode *.go*() détermine alors le nombre de 'pas' (*stepNb*) que le déplacement implique (code 3.3.7) :

```
913     if(motion === "move"){
914
915       stepNb = 1;
916       statementCode = "m";
917
918     } else if(motion === "jump") {
919
920       stepNb = 2;
921       statementCode = "j";
922
923     } else {
924
925       return;
926
927     }
```

Code 3.3.7 : détermination par la méthode *.go()* du nombre de pas à effectuer

En mode 'action', après avoir vérifié que le guide est actuellement toujours sur une zone sécure (attribut *.onTrack* vrai), la méthode *.go*() s'assure que le prochain déplacement à exécuter ne va pas entraîner le guide en dehors de la grille de cellules (bordure du glacier), sous un paroi de laquelle peuvent tomber des rochers (code 3.3.8). Si futur déplacement n'entraîne pas une sortie de la grille, la méthode place dans la variable *nextLocationCode* le symbole du contenu (crevasse, touriste, neige, piste) de la case (figure 3.3.4) à laquelle le déplacement aboutira (code 3.3.8, ligne 1018).

Si le déplacement devait au contraire amener le guide au-delà du glacier (grille), le bruit d'une chute de pierre est joué. La fonction réalise le déplacement sur le topo en déterminant, à partir de la position actuelle du guide (this.*x* et this.*y*) et la nature de son mouvement, le point de chute de la pierre tombant de la paroi, à savoir la cellule dans laquelle le guide va se heurter à la paroi et la pierre va lui tomber dessus (code 3.3.9, lignes 1021ss).

Dans le cas d'une simple avancée (*move*), il s'agit de la cellule où se trouve déjà le guide. On place donc dans la cellule correspondante le symbole 's' pour y remplacer l'image du guide par celle d'une pierre (stone) (lignes 1029 à 1031). Dans le cas d'un saut (jump) (lignes 1033 à 1051), il s'agit de savoir si le guide se trouve, avant d'effectuer le saut, sur une cellule en bordure de glacier ou, sinon, sur une cellule intérieure adjacente à une cellule en bordure de glacier.

```
996     if(this.playing){
997       // B. move guide (one step sequence) physically
998       // on map (playing animation))
999
1000      if(this.onTrack){
1001
1002        if( this.x + this.step.x * stepNb < this.roadMap[0].length
1003          // guide not moving out of map on right side
1004
1005          && this.x + this.step.x * stepNb >= 0
1006          // guide not moving out of map on left side
1007
1008          && this.y + this.step.y * stepNb < this.roadMap.length
1009          // guide not moving out of map on down side
1010
1011          && this.y + this.step.y * stepNb >= 0){
1012          // guide not moving out of map on up  side
1013
1014          nextX = this.x + this.step.x * stepNb;
1015          nextY = this.y + this.step.y * stepNb;
1016          // new guide route coordinates after step done
1017
1018          nextLocationCode = world[nextY][nextX];
1019          // get ground code of new guide location (safe, unsafe, crack)
1020
1021        } else {
```

Code 3.3.8 : vérification d'absence de débordement par la méthode *.go*()

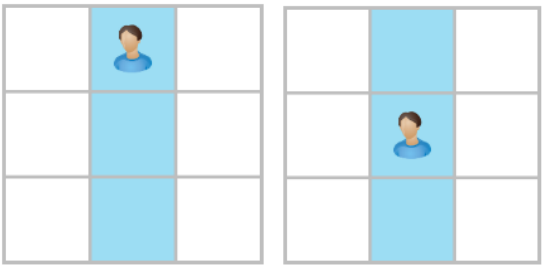


Figure 3.3.12 : position du guide préalable à un saut (cellule en bordure ou adjacente)

Si le guide est déjà en bordure du glacier (figure 3.3.12, gauche), son saut est avorté et provoque une chute de pierre sur la cellule en bordure. L'image du guide ('p') est simplement remplacée par celle de la pierre ('s') (code 3.3.9, ligne 1039). Si le guide se trouve sur une cellule adjacente à une cellule de bordure (figure 3.3.12, droite), son saut se termine prématurément sur la cellule en bordure. L'image du guide est retirée de la cellule de sa position initiale (ligne 1044), cette cellule prenant à nouveau la forme d'une simple case bleue représentant une zone sécure vide ('x'). Le guide n'effectue qu'un demi-saut aboutissant dans la cellule adjacente, ses coordonnées étant modifiées en conséquence (lignes 1045 et 1046). Une pierre est placée dans cette cellule adjacente ('s', ligne 1047) (figure 3.3.13).

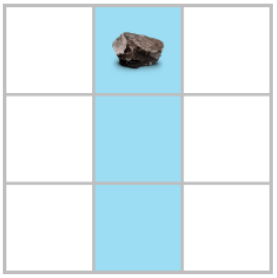


Figure 3.3.13 : chute de pierre après un saut ou une avancée menant hors du glacier (paroi)

```
1021    } else {
1022      // guide moves out of map borders
1023      // process out map border motion in case of move and jump
1024
1025         fallingRocksSound.play();
1026
1027         switch(motion){
1028
1029            case "move" : this.roadMap[this.y][this.x] = "s";
1030                          // replace guide by falling stone after moving
1031                          break;
1032
1033            case "jump" : if(this.x + this.step.x > this.roadMap[0].length - 1
1034                             || this.x + this.step.x < 0
1035                             || this.y + this.step.y > this.roadMap.length - 1
1036                             || this.y + this.step.y < 0){
1037                            // guide jumping from a glacier border cell
1038
1039                             this.roadMap[this.y][this.x] = "s";
1040
1041                          } else {
1042                            // guide jumping from 1 step away glacier border cell
1043
1044                            this.roadMap[this.y][this.x] = "x";
1045                            this.x = this.x + this.step.x;
1046                            this.y = this.y + this.step.y;
1047                            this.roadMap[this.y][this.x] = "s";
1048
1049                          };
1050
1051                          break;
1052
1053         }
1054
1055         this.hit = true;
1056         this.missionFailed = true;
1057
1058         return;
1059         // stone drawn on map, exit game
1060
1061       nextLocationCode = "-";
1062
1063    } // end of processing guide's moving out of border
```

Code 3.3.9 : affichage de la pierre tombée de la paroi suite à une avancée ou un saut (*.go*())

Une fois la pierre affichée et le guide disparu, l'attribut *.missionFailded* passe de *false* à *true* et le contrôle est retourné à la méthode *.playNext*(), qui mettra un terme à l'animation et réinitialisera la grille dans son état d'origine, avant déplacement du guide, afin de permettre au joueur d'améliorer son code et de procéder à une nouvelle tentative.

Après s'être occupé du cas d'un débordement de scène ou avoir déterminé les coordonnées de la cellule de destination du guide suite à son déplacement ainsi que fait une copie de symbole de l'image occupant cette cellule avant déplacement, la fonction *.go*() se préoccupe de savoir si la cellule de destination est sûre (cellule bleue) ou dangereuse (crevasse, glacier recouvert de neige) (code 3.3.10).

```
1066    if(nextLocationCode !== "x"
1067       && nextLocationCode !== "t"
1068       && nextLocationCode !== "f"){
1069       // next guide's destination cell is not safe or target
```

Code 3.3.10 : vérification de la dangerosité de la cellule de destination (*.go*())

Si cette dernière n'est pas sûre, le guide tombe dans la crevasse de la cellule de destination. Cette chute se fait en trois étapes, les deux premières étant une séquence d'animation constituée des deux images (figure 3.3.14).

Figure 3.3.14 : images de la séquence d'animation de la chute

Au début de la première étape (code 3.3.11), le guide se trouve encore sur la cellule sûre à partir de laquelle il va entamer son déplacement (attribut *.onTrack* à true). L'image du guide est enlevée de cette cellule de couleur bleue. Son symbole ('p') est remplacé par celui d'une cellule sécure inoccupée ('x') (ligne 1077). Les coordonnées du guide sont modifiées et remplacées par celles de la cellule de destination (lignes 1080 et 1081). Le symbole du guide ('p') concaténé à celui de la crevasse ('c') sont placés dans cette dernière afin d'y afficher l'image du guide sur celle de la crevasse (ligne 1084). L'attribut *.onTrack* est mis à *false* afin de prendre note que le guide ne se trouve plus sur une zone sécure du glacier et préparer la prochaine étape de l'animation.

Afin de répéter la dernière l'opération 'avancer' ou 'bouger' de la liste d'instructions *statements*[ ] de l'itinéraire, le numéro de la prochaine instruction à exécuter est décrémenté

(ligne 1090) pour revenir au numéro de l'instruction provoquant la chute et répéter son exécution à la prochaine séquence d'animation lancée par la méthode .*playNext*(). Le son de la chute dans la crevasse est joué (ligne 1093) et le contrôle est retourné à la fonction .*playNext*().

```
1071      if(this.onTrack){
1072        // #1 opening guide falling into crack scene (1st sequence)
1073        // guide is still on safe cell before moving
1074        // so move guide on dangerous cell (crack or covered crack)
1075        //  on virtual map but guide not falling immediately into crack
1076
1077        this.roadMap[this.y][this.x] = "x";
1078        // replace guide with empty safe blue area in future previous cell
1079
1080        this.x = nextX;
1081        this.y = nextY;
1082        // move guide to new cell setting his new coordinates
1083
1084        this.roadMap[this.y][this.x] = "cp";
1085        // set crack and guide code on destination cell
1086
1087        this.onTrack = false;
1088        // guide is now on unsafe cell over a crack but is still fallen
1089
1090        this.seqNb--;
1091        // previous safe sequence number
1092
1093        fallingBodySound.play();
1094
1095        return;
1096        // guide over crack codes marked on map for current cell
1097
1098      } else {
```

Code 3.3.11 : première étape de la chute du guide dans la crevasse (.*go*())

A la deuxième étape de la chute, lors du prochain appel de la méthode .*go*() par la fonction .*playNext*(), l'attribut .*onTrack* est à *false*. tout comme la valeur de l'attribut .*fell*, signalant que le guide n'a pas encore disparu dans la crevasse qui s'est ouverte sous ses pieds. Si tel est le cas (code 3.3.12), le symbole de l'image du guide et de la crevasse ('cp') est remplacé par celui de la crevasse seule ('c') : le guide va disparaître de la cellule dans laquelle il se trouve et on n'y verra plus que la crevasse dans laquelle il est tombé (ligne 1107).

La chute du guide dans la crevasse est noté dans l'attribut .*fell* qui passe à *true*, préparant ainsi, avec la décrémentation du numéro de la prochaine instruction de déplacement à exécuter à la prochaine séquence d'animation, la réalisation de la troisième et dernière étape de l'animation de la chute. A cet effet, le contrôle est redonné à la méthode .*playNext*().

```
1098      } else {
1099        // guide falls into crack
1100
1101         if(! this.fell){
1102         // #2 guide falling into crack scene (guide falling into crack)
1103         // guide fall into crack marked on map
1104         // guide code removed from current cell on map
1105         // and no more visible
1106
1107           this.roadMap[this.y][this.x] = "c";
1108           this.fell = true;
1109
1110           this.seqNb--;
1111           // previous safe sequence number
1112
1113           return;
1114
1115         } else {
```

Code 3.3.12 : deuxième étape de la chute du guide dans la crevasse (.*go*())

Lors de la troisième étape (code 3.3.13), l'animation est stoppée (lignes 1118 et 1119), la tentative est notée comme ayant échoué (ligne 1120) et l'affichage de la scène (topo du glacier) est réinitialisé, ce qui ramène le guide à sa position initiale et rebouche la crevasse (ligne 1121).

```
1115         } else {
1116           // #3 ending falling guide sequence
1117
1118           noLoop();
1119           this.playing = false;
1120           this.missionFailed = true;
1121           setTimeout(reinitializeWorld, 3000);
1122
1123           return;
1124           // cracked alone drawn on map, exit game
1125
1126         }
1127
1128       }
1129
1130     }
```

Code 3.3.13 : troisième étape de la chute du guide dans la crevasse (.*go*())

Si le déplacement du guide n'entraîne aucune chute dans une crevasse ou chute de pierre provoquant la fin de sa tentative (code 3.3.14), le guide est déplacé dans la cellule à laquelle aboutit son déplacement (lignes 1135 à 1139) après avoir quitté sa cellule de départ, redevenant une zone sécure (cellule bleue) (ligne 1132).

```
1132    this.roadMap[this.y][this.x] = "x";
1133    // replace guide with empty safe blue area in future previous cell
1134
1135    this.x = nextX;
1136    this.y = nextY;
1137    // move guide to new cell setting his new coordinates
1138
1139    this.roadMap[this.y][this.x] = "p";
1140    // set guide code on destination cell
1141
```

Code 3.3.14 : déplacement du guide dans une zone sûre (*.go*())

```
1142        if(nextLocationCode === "t" || nextLocationCode === "f"){
1143        // next cell is target cell (tourist)
1144
1145            this.roadMap[this.y][this.x] += nextLocationCode;
1146            // add target cell code to guide code
1147            // to display both corresponding icons
1148
1149            this.missionSuccess = true;
1150            data[worldNb]["succeed"] = true;
1151            // set success flags to true for this world
1152
1153            paintWorldIcon(worldNb + 1, "#069137");
1154            // pain in green world icon number
1155
1156            setTimeout(goToNextWorld, 4000);
1157
1158        }
1159
1160    }
1161
1162 }
```

Code 3.3.15 : mission accomplie (*.go*())

Enfin, si dans la cellule de destination se trouve la touriste à secourir ('t')[3], le symbole de l'image de cette dernière est ajoutée dans cette cellule à celle du guide ('pt') et les deux personnages y seront alors affichés (code 3.3.15). La mission est notée comme réussie du côté du joueur (attribut *.missionSuccess* à *true*) et dans les données du topo courant (attribut *.succeed* noté *true*). Sous la grille de jeu, l'icône numérotée du monde correspondant est coloriée en vert. La fonction *goToNextWorld*() est alors appelée dans un intervalle de 4 secondes (le temps que l'accompagnement sonore du succès s'achève) afin d'afficher le prochain topo non encore résolu.

La fonction *goToNextWorld*() (code 3.3.16) appelle la fonction *allSoundsFadeOut*() afin de réduire progressivement le volume de toutes les illustrations sonores (à l'exception du fond sonore constitué du bruit du vent). Elle détermine ensuite le numéro du prochain topo (*nextWorldNb*) non encore résolu en faisant appel à la fonction *getNextUnsolvedWorldNb*(), en précisant à cette dernière à partir de quel topo effectuer sa recherche, à savoir le topo suivant le topo venant d'être résolu.

Si la fonction *getNextUnsolvedWorldNb*() ne trouve aucun topo non résolu jusqu'à la fin de l'ensemble des topos, elle poursuit sa recherche au début de cet ensemble. Si tous les topos ont été résolus, cette fonction retourne la valeur -1. Dans ce cas, la partie est terminée, une musique de succès est jouée et la fonction *blinkWorldIcons*() est appelée afin de faire clignoter toutes les icônes numérotées des topos. Dans le cas contraire, l'icône du prochain topo à résoudre est activée (peinte en orange) et la fonction *goToWorld*() appelée afin de passer à ce topo (affichage du topo et des codes visuel et littéral déjà composés pour ce topo).

```
1991 function goToNextWorld(){
1992
1993   var nextWorldNb;
1994
1995   allSoundsFadeOut(2);
1996
1997   nextWorldNb = getNextUnsolvedWorldNb(worldNb + 1);
1998
1999   if(nextWorldNb == -1){
2000
2001     if(endMissionSound) endMissionSound.play();
2002     blinkWorldIcons(0, 500);
2003
2004   } else {
2005
2006       paintWorldIcon(nextWorldNb, "#069137");
2007       goToWorld(nextWorldNb);
2008
2009   }
2010
2011 }
```

Code 3.3.16 : fonction *goToNextWorld*()

[3] ou un drapeau de jalon (étape) ('f')

3.4 tutoriel

Un tutoriel aide l'utilisateur à réaliser un exercice en lui permettant de découvrir les principales fonctionnalités de l'application. Il est constitué d'une ou de plusieurs séquences de dias. Une séquence de dias est toujours attachée à un topo d'un exercice. Tout exercice ne dispose pas obligatoirement d'un tutoriel. Ce dernier est une extension venant se greffer sur un exercice afin de le compléter. Lorsqu'un exercice dispose d'un tutoriel, une référence à ce dernier figure dans les informations du fichier *.json* de l'exercice. La valeur de l'attribut *tutorial* des informations de l'exercice correspond à la fois au nom du tutoriel et du dossier le contenant.

Ainsi, par exemple, l'exercice de 'prise en main' dispose d'un tutoriel dont le nom ainsi que celui du dossier le contenant est 'introduction' (code 3.4.1, ligne 6). L'attribut *activeTutorial* précise si le tutoriel doit être affiché d'emblé à l'ouverture de l'exercice (*true*) ou si une action de l'utilisateur est requise afin de l'afficher à la demande, si nécessaire pour ce dernier.

```
1 {
2   "info": {
3     "name": "Sauvez Julie - prise en main",
4     "author": "Collège St-Michel",
5     "currentItem": 0,
6     "tutorial": "introduction",
7     "activeTutorial": true
8   },
```

Code 3.4.1 : en-tête de l'exercice de prise en main et référence à son tutoriel

On trouve dans le dossier d'un tutoriel l'ensemble de ses dias regroupés dans un objet Javascript défini au sein d'un fichier au format *JSON*. Ce fichier doit avoir, par définition, le même nom que le dossier du tutoriel dans lequel il se trouve et vers lequel pointe le fichier d'exercice correspondant. Le dossier du tutoriel peut contenir un dossier 'images' ainsi qu'un dossier 'videos'. On trouve dans ceux-ci les images et vidéos à afficher sur les dias de présentation du tutoriel (figure 3.4.1).

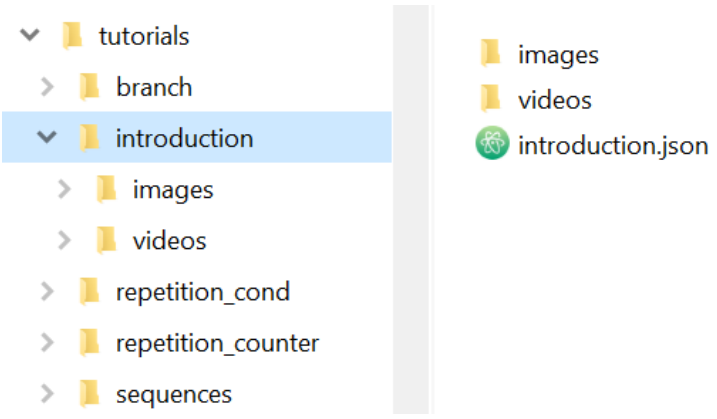


Figure 3.4.1 : arborescence des dossiers d'un tutoriel

Un objet de classe *Tutorial* est constitué de deux objets-membres imbriqués, le premier placé dans son attribut *info* (constituant son en-tête), le second dans son attribut *content* (représentant son corps). Les attributs de l'objet *info* contiennent des informations sur le tutoriel : nom, auteur, numéro de la scène (*currentStageNb*) ainsi que celui de la dia (*currentSlideNb*) active (code 3.4.2). La scène est une séquence de dias attachée à un topo de l'exercice.

```
1  {
2    "info": {
3      "name": "introduction",
4      "author": "Collège St-Michel",
5      "currentStageNb": 0,
6      "currentSlideNb": 0
7    },
```

Code 3.4.2 : en-tête (info) du tutoriel de prise en main

Le contenu du tutoriel (objet *content*) est constitué d'un attribut *stage* (scènes) (code 3.4.3, ligne 11). On trouve dans celui-ci une liste de scènes, chaque scène étant constituée de différents attributs (lignes 17 à 20) : numéro du topo auquel la scène est attachée (*number*), titre de la scène (*title*), visibilité par défaut (*visible*) et son état (*viewed*, précisant si la scène a déjà été visionnée ou non).

```
 9    "content": {
10
11      "stage" :
12
13        [
14
15          {
16
17            "number" : 0,
18            "title": "Sauvez Julie",
19            "visible": true,
20            "viewed": false,
21            "slide" : [
```

Code 3.4.3 : liste de scènes du tutoriel d'introduction et en-tête de la première scène

A ces attributs d'une scène s'ajoute l'attribut 'slide' (ligne 21) dans lequel on trouve la liste des dias d'une scène (code 3.4.4). Chaque dia a un nom (*name*, affiché comme titre de la dia), un contenu texte (*text*) ainsi que, optionnellement, une image ou une vidéo) (*image*,

video). *Image* et *vidéo* sont des objets ayant deux attributs. L'attribut *src* précise le nom du fichier image ou vidéo à afficher, localisé dans le sous-dossier 'images' ou 'vidéos' du tutoriel alors que *size* contient la valeur de la propriété *CSS* 'size' appliquée à l'image ou la vidéo.

```
21            "slide" : [
22
23              { "name": "Introduction",
24                "text": "Julie s'est imprudemment ave
25                "image": {"src": "julie128x128.png",
26                          "size": "128px"
27                }
28
29              },
30
31
32              { "name": "Mission",
33                "text": "Aide Marc à rejoindre Julie
34                "image": {"src": "topo - intro.png",
35                           "size": "35vh"
36                }
37
38              },
```

Code 3.4.4 : liste des dias de la scène 0 du tutoriel d'introduction (2 premières dias)

Figure 3.4.2 : 2ème dia de la première scène du tutoriel d'introduction

Enfin, les flèches au bas de chaque dia permettent à l'utilisateur de passer à la dia suivante ou de revenir à la dia précédente (figure 3.4.2).

4. Conclusion

> "En voulant moins, peut-être aurait-il obtenu davantage"
> Honoré de Balzac (1836), *Le lys dans la vallée*

L'enseignement et l'apprentissage de la programmation sont tout sauf une sinécure. Ils demandent un important engagement, de la discipline et de la régularité. Nombreux sont les enseignants souhaitant transmettre à travers leurs cours d'informatique leur enthousiasme pour la programmation mais devant déchanter malgré toute leur bonne volonté et le temps qu'ils y consacrent. Comme le souligne Mathias Hauswirth (2022) du Département d'informatique de l'Università della Svizzera italiana, "learning to Program is Hard" et les écueils dans cet apprentissage sont nombreux. Le problème a priori élémentaire des précipitations ('rainfall problem') de Soloway (1986) a posé paradoxalement de grandes difficultés de par le monde à des générations d'étudiants en informatique en fin de première année universitaire sortant d'un cours d'introduction à la programmation. Il constitue aujourd'hui encore une jauge pour les chercheurs afin d'évaluer les cours de programmation. Soloway a énoncé ainsi ce problème :

> *Ecrire un programme lisant une suite de nombres représentant le niveau quotidien de chutes de pluie jusqu'au nombre 99999 en ignorant les valeurs négatives et calculer puis afficher la moyenne quotidienne des chutes de pluie données en entrée.*

La première fois que ce problème – a priori trivial pour un informaticien – a été soumis en 1982 à des étudiants en informatique de l'Université de Yale en fin de première année, le résultat obtenu a fortement surpris : seuls 14% des étudiants ont réussi à le résoudre correctement… Les résultats ne sont guère meilleurs et restent insuffisants avec des étudiants de deuxième année : 36%. Depuis, aussi bien à Yale que dans d'autres universités, ce test – ou des légères variantes de celui-ci – a régulièrement été donné en évaluation à l'issue de cours de programmation de première année. Et de manière encore plus surprenante, les résultats n'étaient guère meilleurs. Seppälä et al (2015), réalisant une synthèse des nombreuses évaluations et recherches faites sur le problème des précipitations, font état des conclusions pessimistes de leurs collègues à ce sujet, à l'instar de celle de Simon (2013) ou de Taylor et al. (2014) :

> "The general conclusion is that large numbers of students are still making the same sorts of error that they were making 30 years ago." Simon (2013)

> "Students struggle with this problem today as they did in 1982." Taylor et al. (2014)

Cette évaluation, répétée pendant des décennies, l'a été dans des conditions pouvant varier d'un cours à un autre (examen écrit, examen sur ordinateur, contrôle continu ou examen de fin de cours, examen openbook ou non, langage de programmation utilisé, etc). Il n'empêche que les résultats obtenus sont restés particulièrement faibles. Par exemple, une étude

portant sur 5 universités dans 4 pays différents aboutit à un taux de succès de 21% (McCracken et al., 2001).

Lister et al. (2004) jugent le problème de Soloway trop complexe pour des étudiants de première année, sa résolution nécessitant la combinaison de nombreux concepts et de schèmes de programmation pour un débutant. En outre, Lister émet l'hypothèse selon laquelle les étudiants peinent à écrire le code de Soloway de par des lacunes en interprétation de leur propre code. Afin d'évaluer les compétences acquises par des étudiants de première année, Lister et ses collègues focalisent sur les aptitudes des étudiants à interpréter des codes donnés résolvant chacun un problème élémentaire unique de programmation. Pour cela, ils soumettent les étudiants à deux types de problèmes, dont les réponses sont à choisir parmi celles proposées dans un questionnaire à choix multiples. Ils doivent tout d'abord trouver les données générées par l'exécution de différents codes résolvant chacun un problème élémentaire de programmation. Ils choisissent ensuite le code manquant de codes à compléter dont on connaît l'objectif.

Les résultats obtenus sont alors nettement meilleurs qu'avec le test de Soloway, mais restent décevants : seuls 65% des étudiants réussissent le test. Ce d'autant plus que tous les étudiants parvenant à prédire mécaniquement les données produites par l'exécution d'un code ne sont pas à même de comprendre le sens du code (en répondant à la question 'à quoi sert ce code ?') : ils en donnent une interprétation descriptive et non pas fonctionnelle. Dans une autre étude menée par Lister et al. (2006), il apparaît que seuls un tiers des étudiants interrogés sont à même de donner une interprétation fonctionnelle de codes élémentaires.

Plus de 10 ans après, McCracken et ses collègues (2013, cité par Hauswirth, 2022) obtiennent cependant de meilleurs résultats au problème de Soloway, dans une étude portant sur 12 universités de 10 pays avec des taux de réussite allant de 61 à 75%. Cependant, dans ce cas, les étudiants avaient été préparés au préalable en s'exerçant sur des exercices de nature structurellement similaire à celui des précipitations. Des groupes témoins d'étudiants n'ayant pas été drillés au préalable n'obtiennent en revanche, toujours dans cette même étude, que des taux de réussite allant de 12 à 32%...

Cependant, d'autres auteurs remettent en question ces résultats fatalistes, arguant qu'avec les années les cours de programmation s'améliorent de par l'expérience acquise par les enseignants. Ainsi, Seppäla (2015) et ses collègues finlandais ont obtenu de biens meilleurs résultats avec leurs étudiants de première année au problème des précipitations, dans 3 cours donnés au sein de différentes hautes écoles finlandaises (université, école d'ingénieurs), avec respectivement 45%, 53% et 72% de réussite, ce qui contraste fortement avec les maigres résultats des études précédentes.

Néanmoins, ce n'est que dans le premier cours que les étudiants ont été évalués dans des conditions strictes d'examen (examen en salle de cours sur ordinateurs de l'école, sans connexion Internet ni accès aux ressources du cours, temps limité). L'examen prenait la forme d'un exercice noté facultatif, comptant pour la note finale et récompensé par un billet de cinéma gratuit. Les étudiants avaient également réalisé au préalable de nombreux exercices sur les différents schèmes d'algorithmes nécessaires à la résolution du problème des précipitations (saisie en boucle de nombres, calcul de la moyenne d'une série de

nombres, etc).. Dans les deux autres cours, l'évaluation était une évaluation continue avec des devoirs à rendre de leçon en leçon, évalués et comptant pour la note finale. Un de ces devoirs était le problème des précipitations. Les étudiants avaient accès à toutes les ressources souhaitées (Internet, support de cours, etc) et pouvaient travailler en groupe de deux. Ils avaient également des séances d'exercices avec l'aide d'assistants. Les étudiants du troisième cours disposaient au sein de leur support de cours d'exemples de codes pour les problèmes fondamentaux étudiés, codes qui pouvaient être copiés / collés et adaptés lors de la réalisation d'exercices.

Seppäla et ses collègues admettent que les conditions n'étaient pas similaires à celles d'autres études, qui elles-mêmes varient entre elles. Plus les conditions de résolution du problème étaient souples, plus les étudiants sont parvenus à résoudre le problème. Mais on peut s'interroger dans quelle mesure les étudiants du 2ème et 3ème cours ayant retourné des solutions correctes ont compris ces dernières et dans combien de cas ils en étaient effectivement les auteurs. Cette dernière interrogations se poserait d'autant plus aujourd'hui alors que l'on retrouve la solution du problème des précipitations et de nombreux autres problèmes classiques directement sur Internet, exprimées dans une multitude de langages de programmation, à l'instar de celles que l'on trouve sur ce site : https://rosettacode.org[4] Et cela, sans compter les solutions fournies par des intelligences artificielles telles que ChatGpt.

Au final, la conclusion de Soloway et de la majorité des recherches effectuées depuis plus de 30 ans sur le sujet tendent à rester pertinente : placé derrière un ordinateur ou une feuille de papier, sans autre ressources et disposant d'une limite de temps imparti, un étudiant en informatique, sans expérience et connaissance préalable du sujet, sortant d'un cours d'introduction à la programmation de première année universitaire, aura beaucoup de peine à résoudre par lui-même le problème, a priori trivial, des précipitations de Soloway, et ce, indépendamment du langage de programmation littéral utilisé. Selon Simon (2013), ce problème est même devenu plus difficile à résoudre dans les environnements de programmation contemporains, plus complexes qu'une simple console, introduisant de nouveaux concepts à maîtriser tels que les événements ou l'asynchronie.

Comment expliquer dès lors un tel paradoxe ? Guzdial (2011) avance plusieurs hypothèses, validées par différentes recherches qu'il cite dans son article : 'Why is it so hard to progam?' Pour Guzdial, ce ne sont pas tant les problèmes algorithmiques à résoudre qui posent problème, des enfants étant à même de concevoir un algorithme de tri par exemple, mais l'expression de leur solution en suivant les règles strictes d'un langage de programmation s'adressant à une machine et respectant les contraintes architecturales de cette dernière. Et Guzdial de citer différentes études montrant que l'expression des solutions de problèmes algorithmiques dans un langage naturel pose sensiblement moins de problème. Une majorité d'étudiants débutant en programmation est par exemple à même de résoudre des problèmes réputés complexes pour leur niveau en les formulant dans un langage naturel, suivant une logique plus intuitive et proche du raisonnement humain, à l'instar de problèmes de programmation concurrente (Lewandowski, 2007).

Ainsi, pour Guzdial, les langages littéraux utilisés pour l'initiation à la programmation devraient être plus proches du langage naturel. Les langages visuels vont-ils en ce sens,

[4] https://rosettacode.org/wiki/Soloway%27s_recurring_rainfall

levant la problématique du respect d'une syntaxe trop rigide et peu naturelle ? Oui, selon Guzdial, qui met cependant en garde contre une limite a priori surprenante : s'il est plus aisé pour un débutant de coder visuellement que littéralement, il est en revanche plus difficile pour ce même débutant d'interpréter un code visuel qu'un code littéral (Moher, 1993).

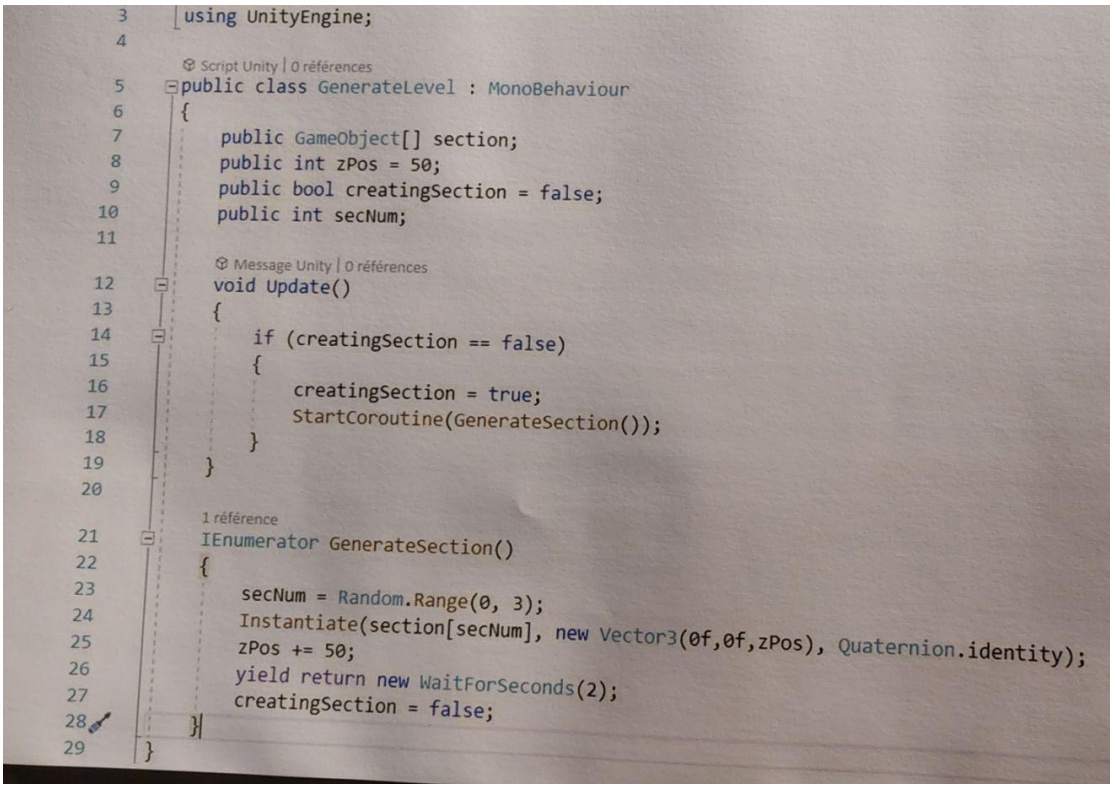
Toujours selon Guzdial, placer l'apprentissage des concepts de programmation dans un contexte motivant pour l'étudiant a un impact important sur l'efficacité de son apprentissage de la programmation. Il mentionne l'expérience faite au Georgia Institute of Technology où un cours d'introduction à l'informatique a été imposé à tous les étudiants de premier cycle dès 1999, quelle que soit la discipline étudiée. Les résultats obtenus se sont significativement améliorés lorsque l'on est passé d'un cours conventionnel avec une approche plutôt théorique à un cours où les concepts élémentaires de programmation sont étudiés à travers leur mise en application dans la résolution de problèmes pratiques auxquels les étudiants sont confrontés dans leur quotidien numérique, à savoir, dans le cas présent, la manipulation de ressources multimédia. Ainsi, par exemple, l'itération et la manipulation d'un tableau de nombres portera sur la transformation des pixels d'une image bitmap couleur en une image à niveaux de gris ou la suppression des yeux rouges. Guzdial souligne que si, globalement, l'impact de cette contextualisation améliore les résultats des étudiants, elle a sensiblement accru ceux des étudiants ayant le plus de peine avec un cours classique.

Guzdial fait ici référence à différentes expériences et études confirmant les conséquences positives de l'ancrage d'un cours de programmation dans le vécu des élèves afin de susciter leur motivation. Ecrire un programme affichant dans une console une suite de Fibonacci ou les nombres premiers entre 1 et 100 ne génère vraisemblablement pas le même intérêt qu'afficher en surimpression d'une vidéo les paroles d'une chanson pour un karaoké ou afficher sur une page web la liste des publications des amis d'un utilisateur. On peut s'étonner parfois du manque de pragmatisme que l'on trouve dans des cours de programmation conventionnels dont les exemples illustratifs portent souvent sur des applications de notions mathématiques. Cela n'est cependant guère surprenant. Historiquement, les premiers professeurs d'informatique dans les hautes écoles n'étaient pas informaticiens, cette discipline n'existant pas encore. Son développement et son enseignement ont été le plus souvent l'initiative de mathématiciens, leur discipline d'origine ayant fortement influencé le contenu de leurs cours, et ce, de génération en génération. Cette influence reste encore très présente dans des cours d'informatique continuant d'être théoriques, basés sur une approche déductive plutôt qu'inductive, rendant d'autant plus difficile une contextualisation de leur contenu ainsi qu'une approche constructiviste piagétienne reposant sur un apprentissage par la pratique au sein du monde environnant.

Fort heureusement le pourcentage d'étudiants réussissant aux cours de premier cycle d'introduction à la programmation est plus élevé que le taux de réussite au test des précipitations de Soloway. Selon une étude de Watson et Li (2014) ayant analysé les résultats de 161 cours d'introduction à la programmation dispensés dans les hautes écoles de 15 pays différents, en moyenne, 67.7% des étudiants les ayant suivis passent la rampe. Ce qui signifie tout de même qu'un tiers des étudiants échouent à l'évaluation. Un taux qui, pour Andrew Luxton-Reilly (2016) de l'Université d'Auckland, ne serait cependant pas extraordinaire dans un cours d'introduction de première année universitaire, quelle que soit la discipline y étant enseignée.

On peut tout de même s'interroger quant à savoir si ce taux de réussite n'est pas quelque peu surestimé, omettant un phénomène que l'on pourrait qualifier d' 'évaluation de complaisance', lorsque les critères d'évaluation se révèlent insuffisants, permettant à des étudiants dont les aptitudes développées au cours sont manifestement elles-mêmes insuffisantes, de tout de même le réussir. Lors de l'évaluation des travaux, l'évaluateur peut également appliquer de manière très vague des critères adéquatement spécifiés, interprétant comme étant corrects des éléments de réponses qui ne le sont pas ou insuffisamment. De telles surévaluations, conscientes ou non, peuvent avoir comme motivation de dissimuler un taux d'échec effectif élevé, de prévenir les protestations qui ne manqueraient pas de s'en suivre chez les étudiants ou d'éviter une mauvaise évaluation du cours par ces derniers.

L'illusion de compétences peut également biaiser une évaluation lorsque l'on s'imagine que la majorité des étudiants d'un cours y ont acquis des compétences qu'ils n'ont pas en réalité, celles-ci étant surestimées. A l'instar de celles d'élèves de 8-9 ans suivant un cours facultatif de développement de jeux vidéos en 3 dimensions, supposés comprendre et adapter un code en langage C# faisant appel à des notions avancées de programmation et de mathématiques, alors même que ces enfants en sont au stade de l'apprentissage des livrets de multiplications. La figure 4.1 est une photographie d'une des affiches présentée lors d'une exposition du résultat du travail accompli par les élèves ayant suivi un tel cours. On peut se douter que ce code, autogénéré par l'IDE Unity, n'a pas été écrit et ni même compris par des enfants de cet âge, même si cela peut donner aux parents l'illusion que c'est le cas.



```
3   using UnityEngine;
4
    Script Unity | 0 références
5   public class GenerateLevel : MonoBehaviour
6   {
7       public GameObject[] section;
8       public int zPos = 50;
9       public bool creatingSection = false;
10      public int secNum;
11
        Message Unity | 0 références
12      void Update()
13      {
14          if (creatingSection == false)
15          {
16              creatingSection = true;
17              StartCoroutine(GenerateSection());
18          }
19      }
20
        1 référence
21      IEnumerator GenerateSection()
22      {
23          secNum = Random.Range(0, 3);
24          Instantiate(section[secNum], new Vector3(0f,0f,zPos), Quaternion.identity);
25          zPos += 50;
26          yield return new WaitForSeconds(2);
27          creatingSection = false;
28      }
29  }
```

Figure 4.1 : code d'un projet de jeu vidéo 3d d'un enfant en classe de 3e année primaire

Au bout du compte, même si la majorité des étudiants suivant un cours de programmation de premier cycle le réussisse, si plus d'un étudiant sur trois n'y parvient tout de même pas, on pourrait en conclure, comme la plupart des chercheurs et professeurs travaillant sur la thématique, que cet apprentissage est une dure épreuve par laquelle les étudiants de premier cycle – qu'ils étudient l'informatique ou d'autres disciplines en ayant besoin – doivent passer. Ni les étudiants ni les professeurs ne pourraient hélas y changer quelque chose et il faudrait voir cette conclusion comme une fatalité : à moins de se bercer d'illusions, 'learning to program is hard'.

Ce n'est pourtant pas ce que pense Andrew Luxton-Reilly ayant, à titre provocateur, intitulé son article présenté en 2016 à l'ACM Conference on Innovation and Technology in Computer Science Education à Arequipa au Pérou : 'Learning to Program is Easy'. Pour ce professeur du Département d'informatique de l'Université d'Auckland, l'apprentissage de la programmation n'est pas intrinsèquement difficile. Pour preuve, cela fait depuis plus de 40 ans que des enfants apprennent avec succès à programmer des ordinateurs à l'aide d'environnements de développement et de bibliothèques conçus à cet effet, avec des langages accessibles à des débutant, de *Logo* à *Scratch*. Selon Luxton-Reilly, ce ne sont pas non plus intrinsèquement les cours d'introduction à la programmation qui sont en cause, nombreux étant de par le monde les professeurs donnant des formations de bonne qualité dans ce domaine.

Si le problème du peu d'efficacité de l'apprentissage des bases de la programmation n'est pas de nature qualitative, quel est-il donc dans ce cas ? Pour Luxton-Reilly, son origine première est fondamentalement quantitative : trop et trop vite. L'apprentissage de la programmation est une activité consommant énormément de temps, passant par de nombreux exercices et autres activités pratiques permettant d'assimiler la vaste étendue de concepts mis en application. Enseignants et professeurs d'informatique sous-estiment très largement ce temps d'apprentissage. Cela engendre des attentes irréalistes des enseignants, matérialisées par des listes d'objectifs et des plans d'études bien trop ambitieux menant à un important taux d'échec et d'abandon. Débordé par le travail qu'on leur demande d'accomplir, nombre d'étudiants décrochent en cours de route, se réfugiant dans une étude superficielle du sujet et le plagiat de solutions d'exercices, tentant de virevolter en espérant passer tant bien que mal l'examen avec un investissement minimal en temps ou comptant sur la compensation d'une note insuffisante par celles obtenues dans d'autres domaines que la programmation.

Ce n'est pas le saut à la perche qui est foncièrement difficile ni les entraîneurs qui ne seraient pas à la hauteur. Mais bien la barre qui serait donc placée bien trop haute, le temps d'entraînement largement insuffisant, le rythme de ce dernier beaucoup trop rapide pour que l'on puisse espérer réussir à passer la rampe. Luxton-Reilly en veut pour preuve les différentes études, dont celles portant sur le problème de Soloway, montrant qu'une année après l'avoir suivi, les étudiants d'un cours d'introduction à la programmation arrivent résoudre des problèmes qu'ils n'étaient pas capables de solutionner en fin de cours. Selon lui, les étudiants ayant déjà des difficultés dans d'autres domaines sont d'autant plus négativement impactés par ces attentes et ces rythmes de travail excessifs. Ce professeur émet également l'hypothèse selon laquelle ces attentes disproportionnées constitueraient un élément d'explication des réticences des femmes face à l'étude de la programmation.

Luxton-Reilly invite les professeurs de programmation à remettre en question leurs attentes en prenant conscience de la surcharge de travail engendrée par celles-ci tout en réduisant la voilure de leurs objectifs ou en étalant leur atteinte sur des périodes de temps plus large.

> *Our current approach to teaching programming is to cover too much content too rapidly and expect students to be able to program at a higher level than they are capable of achieving at the end of an introductory programming course. The expectations we set for our students result in programming courses that are notoriously time-consuming and have high drop out and failure rates. These factors appear to have a greater impact on women and may be partially responsible for the gender inequity observed in the Computer Science discipline. This is not because programming is intrinsically difficult (at least not at novice level), but rather because we have collectively adopted disciplinary norms that are, and always have been, unrealistic. Learning to program is easy — all we need to do is collectively shift our view, and teach to achievable outcomes.*
>
> *Learning to Program is Easy*
> Andrew Luxton-Reilly (2016)

Les études et analyses qui précèdent portent sur l'apprentissage de la programmation par des étudiants de premier cycle universitaire. Qu'en est-il dans les gymnases préparant aux études ? Les attentes des enseignants sont-elles plus réalistes ? Les élèves disposent-ils de plus de temps pour ces apprentissages, qu'ils réalisent à un rythme moins soutenu qu'à un niveau universitaire ? A priori la réponse semble affirmative. Mais on est en droit d'en douter lorsqu'on y regarde de plus près. Un cours d'introduction à la programmation au gymnase porte sur les mêmes éléments et concepts que ceux étudiés dans un cours universitaire. Dans le cas du cours d'introduction à l'informatique devant être obligatoirement suivi par tous les gymnasiens suisses, les élèves ont généralement 3 à 4 ans de moins que les étudiants suivant pareil cours dans les hautes écoles. Et la programmation ne constitue qu'une des thématiques abordées dans ce cours, au côté de la représentation des données, de leur stockage, de leur transmission ou de leur protection. Tous ces sujets sont généralement étudiés dans un cours de 2 leçons de 45 minutes hebdomadaire sur deux ans.

Contrairement à des étudiants se voyant habituellement offrir pour chaque heure de cours une heure d'exercices durant laquelle ils bénéficient de l'aide d'assistants, tel n'est pas le cas pour un gymnasien devant faire ses devoirs d'informatique par lui-même dans un laps de temps fort limité car en ayant également quotidiennement d'autres devoirs à effectuer dans d'autres disciplines, ayant le plus souvent un statut plus important. A la concurrence du temps consacré aux autres disciplines s'ajoutent celle de disciplines artistiques et sportives extrascolaires, celle des activités sociales et, de nos jours, de plus en plus, celle des écrans et du divertissement.

On peut donc légitimement supposer que l'apprentissage de la programmation souffre des mêmes difficultés que dans les hautes écoles, qui plus est dans le cadre de cours obligatoires pour tous. Des attentes trop élevées sont également probables à ce niveau. Lorsqu'un projet de plan d'études d'un cours d'informatique au gymnase est soumis en consultation à des professeurs d'informatique enseignant dans des hautes écoles, l'observation principale est systématiquement la même : "Vous pensez vraiment pouvoir faire

tout cela avec vos élèves ? Cela semble bien ambitieux…" Ce n'est cependant pas pour autant que ces plans d'études subissent des cures d'amaigrissement, bien au contraire, d'aucun voulant encore y ajouter de nouvelles thématiques, à l'instar de l'intelligence artificielle ou des implications du numérique sur la société. Des études devraient également être menées dans les gymnases afin d'y voir plus clair sur cette problématique afin. le cas échéant. de prendre des mesures correctives, en réduisant et recentrant la voilure des plans d'études et / ou en augmentant la dotation-horaire des cours d'informatique.

Au bout du compte, que peut-on retenir de ces réflexions sur la conception d'un moyen d'apprentissage polymorphe de la programmation ? Qu'un tel outil doit être adapté au niveau d'apprentissage de son utilisateur, un langage de programmation visuel étant en adéquation avec le stade piagétien des opérations concrètes mais pouvant également être pertinent comme point de départ pour un élève ayant atteint le stade des opérations formelles. Ce moyen a également pour but de faciliter la transition d'un langage visuel à un langage littéral à l'aide desquelles l'élève compose des programmes dont l'exécution permet la manipulation ou la création d'objets pouvant eux-mêmes être visualisés.

Selon Piaget, c'est à travers la construction de schèmes de manipulation – à savoir des algorithmes opératoires – que l'enfant se construit son savoir. Cela ne surprend dès lors guère que pour un enfant, l'apprentissage de la programmation peut se révéler facile et naturel lorsqu'il dispose d'un outil adéquat. Les objets manipulés devraient idéalement faire partie du contexte pratique de l'enfant ou de l'adolescent afin que celui-ci trouve une motivation dans les activités d'apprentissage proposées. Un jeu, une ressource multimédia (graphique, animation, vidéo, musique, etc), un site web, un réseau social ou un robot sont par exemple autant d'objets concrets pouvant être manipulés à l'aide d'un programme.

Le langage de programmation littéral choisi importe fort peu tant qu'il permet de faire de la programmation impérative peu verbeuse et de manipuler un objet stimulant à partir d'un jeu d'instructions sobre. Le moyen d'apprentissage est à même de visualiser ou d'écrire des programmes dans différents langages de programmation littéraux, ce qui permet à son utilisateur de généraliser les notions de programmation apprises en les transposant dans des langages spécifiques. Ce faisant, l'élève prend conscience qu'un algorithme peut se présenter sous différentes formes, dont les instructions de programmes écrits dans des langages littéraux différents, ces derniers n'étant au final que des simples outils et non des finalité en soi, encore moins des religions pour lesquelles mener une croisade.

Sauvez Julie est un prototype expérimental de moyens d'apprentissage polymorphe de la programmation pour débutant tentant d'intégrer les conclusions de ces différents éléments de réflexion. Si son volet dédié à la programmation visuelle a déjà été testé avec satisfaction durant plusieurs années en classe, celui consacré à la programmation littérale et la transition vers cette dernière devrait encore l'être également. Différents éléments manquent encore à cette application, à l'instar de la possibilité d'exécuter le code écrit dans d'autres langages que Javascript (Python, Java). Le concept de variables pourrait également y être implémenté, par exemple en ajoutant des couches d'information aux topos de glaciers, précisant le degré de la pente et la hauteur de neige que l'on trouve dans chaque case. Relevant ces informations, le guide pourrait alors être appelé à déterminer, à l'aide d'expressions, en quels endroits du glacier il risque de chuter ou d'être emporté dans une

avalanche. La possibilité de créer de nouvelles fonctions pourrait également être proposée au joueur.

Un côté serveur pourrait également être développé afin d'offrir la possibilité au joueur de créer un compte lui permettant d'enregistrer ses exercices dans une base de données et de les partager via une simple URL. Les enseignants pourraient quant à eux y créer des comptes avec lesquels ils ouvriraient des classes dans lesquelles seraient regroupés leurs élèves, afin de pouvoir gérer plus efficacement le suivi des travaux réalisés par ceux-ci.

Si un tel moyen d'apprentissage n'a comme principal objectif que d'initier des débutants à la programmation en manipulant un objet simple à l'aide d'instructions élémentaires, il peut être complété et prolongé par d'autres moyens d'approfondissement partageant la même philosophie pédagogique (figure 4.2). L'usage d'un outil de développement tel que *Scratch* ouvre des perspectives additionnelles au niveau d'un langage visuel d'animation multimédia. Tout comme les bibliothèques multilingues et les IDE desktop ainsi que web de *Processing* offrent la possibilité d'aller plus loins dans la découverte des fondamentaux de la programmation littérale à travers la création et la manipulation de ressources multimédia ou web à l'aide d'un jeu d'instructions accessible à des débutants.

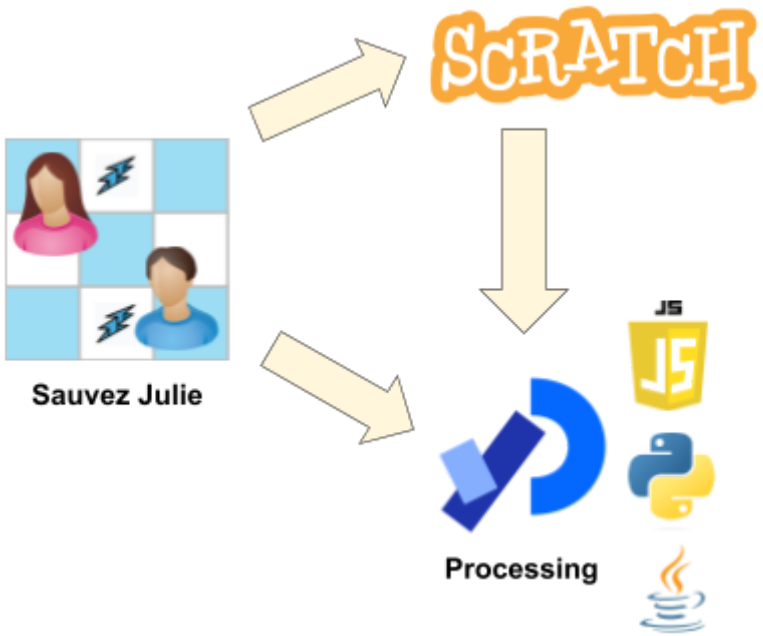


Figure 4.2. : palette de moyens d'apprentissage progressifs de la programmation

Quel avenir à longue échéance pour l'apprentissage de la programmation ? Apprendra-t-on toujours dans 30 ans à programmer avec des langages de haut niveau conventionnels ? On peut déjà observer qu'une couche additionnelle, celle des langages visuels, offre la possibilité de s'en passer. Il n'est cependant pas aisé, pour ne pas dire impossible, de développer des applications complètes et complexes avec de tels langages. Tout d'abord parce que ceux-ci sont principalement conçus pour l'apprentissage des rudiments de la programmation et non pour manipuler des objets complexes de manière performante.

Ensuite parce que la composition et l'édition du code d'un programme de manière visuelle n'est pas très efficace et prend beaucoup de temps en comparaison avec un langage de programmation littérale.

On peut cependant s'interroger, avec le développement de l'intelligence artificielle et l'interprétation du langage naturel (humain) par nos ordinateurs, si une nouvelle couche additionnelle ne pourrait pas s'ajouter à celle des langages de programmation ordinaires. Il serait alors possible d'écrire le code d'un programme dans une langue naturelle quelconque exprimant des algorithmes fondés sur des processus naturels de traitement de l'information reposant sur l'architecture du cerveau humain plutôt que celle artificielle d'un ordinateur. Interpréteurs et compilateurs traduiraient alors ce code source naturel en code écrit dans un langage de programmation de haut niveau ou même directement en langage machine. Comme de nos jours les informaticiens n'écrivent pratiquement plus le code de leurs programmes en langage machine ou en assembleur, les générations à venir n'écriraient alors que rarement le code de leur programme dans les langages de programmation de haut niveau avec lesquelles nous travaillons depuis plusieurs décennies. Si les étudiants en informatique de ces nouvelles générations devraient sans doute encore connaître quelques rudiments de programmation dans ces langages, il serait alors possible que tous les autres – et d'autant plus les élèves de gymnases – ne soient plus appelés à étudier la programmation conventionnelle dans un cours d'introduction à l'informatique, se concentrant sur l'apprentissage de l'algorithmique et de son expression en langage naturel.

* * * * *

Liste des figures

Liste des codes

Liste des tableaux

Références

Bibliographie

Armoni M., Meerbaum-Salant O., Ben-Ari M. (2015), *From Scratch to "Real" Programming* (ACM Transactions on Computing Education)

Bihouix P., Mauvilly K. (2016), *Le désastre de l'école numérique* (Paris: Seuil)

Commissions Romandes de Mathématique, Physique (2018), *Formulaires et tables : Mathématiques, Physique et Chimie* (Granges-Paccot: éditions CRM)

Desmurget M. (2020), *La fabrique du crétin digital* (Paris : Points)

Guzdial M. (2011), *Why is It so Hard to Learn to Program* in 'Making Software: What Really Works, and Why We Believe It' (Sebastopol, CA : O'Reilly)

Hauswirth M. (2022), *Pitfalls in Teaching Programming* (Lugano : Conférence Edu-i-day 2022 au Département d'informatique de l'Università della Svizzera italiana)

Hivert A.-F. (23.05.2023), *La Suède juge les écrans responsables de la baisse de niveau des élèves* (Paris: Le Monde)

Lewandowski G. et al. (2007), *Commonsense computing (episode 3): concurrency and concert tickets* (ICER '07: Proceedings of the third international workshop on Computing education researchSeptember 2007, pp. 133–144)

Lister J., Adams E. S., Fitzgerald S., Fone W., Hamer J., Lindholm M., McCartney R., Moström J. E., Sanders K, Seppälä O., Simon B., Thomas L. (2004), *A multi-national study of reading and tracing skills in novice programmers* (ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education, pp. 119-150)

Lister R., Simon B., Thompson E., Whalley J. L., Prasard C. (2006), *Not seeing the forest for the trees: Novice programmers and the solo taxonomy* (SIGCSE Bull., 38(3):118–122, June 2006).

Luxton-Reilly A. (2016), *Learning to Program is Easy* (ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education. pp. 284-289)

Marry Y., Souillot F. (2022), *La guerre de l'attention* (Paris : L'Echappée)

Madea J. (2001), *Design by Numbers* (Cambridge: The MIT Press)

Mladenović M., Boljat Monika, Žanko Žana (2017), *Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level* (Springer Science+Business Media)

Malan J., Leitner H. (2007), *Scratch for budding computer scientists* (38th SIGCSE technical symposium on Computer science education)

Martínéz-Valdés J. A., Vélazquéz-Iturbidé J. A., Hijon-Néira R. (2017), *A (Relatively) Unsatisfactory Experience of Use of Scratch in CS1* (Cádiz, Spain: Fifth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM'17)

McCraken M., Almstrum V., Diaz D., Guzdial M. J., Agan D., Kolikan Y. B. D., Laxer C. E., Thomas L. A., Utting I., Wilusz T. (2001), *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students* in SIGCSE Bulletin, 33(4): 125-180

McCraken M., Utting I., Tew A. E., Thomas L. A., Bouvier D. J., Frye R., Paterson J. H., Caspersen M. E., Kolikant Y. D. B., Sorva J., Wilusz T. (2013), *A fresh look at novice programmers' performance and their teachers' expectations* in In Proceedings of the 2013 ITiCSE working group reports, ITiCSE -WGR '13, pages 15–32. ACM,

Moher, T. G., Mak D. C., Blummenthal B. (1993). *Comparing the comprehensibility of textual and graphical programs: the case of Petri nets* in C. R. COOK, J. C. SCHOLTZ & J. C. SPOHRER, Eds. 'Empirical Studies of Programmers: Fifth Workshop, pp. 137-161 (Norwood, NJ: Ablex)

Patino B. (2019), *La civilisation du poisson rouge* (Paris : Grasset)

Papert S.. (1981), *Jaillissement de l'esprit : Ordinateurs et apprentissage*, (Paris : Flammarion)

Piaget J., Beth W. (1961), *Etudes d'épistémologie génétique (vol. 14) : épistémologie mathématique et psychologie: essai sur les relations entre la logique formelle et la pensée réelle*, (Paris : Presses Universitaires de France)

Piaget J., Apostel L., Mandelbrot B. (1957), *Etudes d'épistémologie génétique (vol. 2) : Logique et équilibre*, (Paris : Presses Universitaires de France)

Piaget J. (1973), *La psychologie de l'enfant* (Paris : Presses Universitaires De France)

Seppälä O., Ihantola P., Isohanni E., Sorva J., Vihavainen A., (2015), *Do We Know How Difficult the Rainfall Problem is?* (Conference: Proceedings of the 15th Koli Calling Conference on Computing Education Research)

Soloway E. (1986), *Learning to program = learning to construct mechanisms and explanations* (ACM, 29(9):850–858)

Schleicher A. et al. (2015), *Connectés pour apprendre ? les élèves et les nouvelles technologies* (Paris : OCDE)

Toyama K. (2015), *Geek Heresy - Rescuing Social Change from the Cult of Technology* (NYC: PublicAffairs)

Shiffman D. (2012), *The Nature of Code: Simulating Natural Systems with Processing* (New-York : The nature of code)

Simon (2013), *Soloway's Rainfall Problem Has Become Harder* in Computing and Engineering, LaTiCE Conference '13, pages 130–135

Spitzer M. (2019), *Les ravages des écrans - les pathologies à l'ère numérique* (Paris: l'Echappée)

Taylor C., Zingaro D., Porter L., Webb K., Lee C., et Clancy M. (2014) *Computer science concept inventories: Past and future* in Computer Science Education, 24(4):253–276

Watson C., Li F. W (2014) *Failure rates in introductory programming revisited* in 'Proceedings of the 2014 Conference on Innovation' #38; *Technology in Computer Science Education*, ITiCSE '14, pp. 39–44, New York, NY, USA, 2014. ACM.

Whittington K. J., Bills D. P., Hil L. W.I. (2003) *Implementation of alternative pacing in an introductory programming sequence* in Proceedings of the 4th Conference on Information Technology Curriculum, CITC4 '03, pp. 47–53, New York, NY,USA, 2003. ACM.

Webographie

Dubuc B. (consulté le 03.07.2023), *Le développement cognitif selon Jean Piaget* (https://lecerveau.mcgill.ca/flash/i/i_09/i_09_p/i_09_p_dev/i_09_p_dev.html), (Ottawa : Instituts de recherche en santé du Canada)

Ducret J.-J. (consulté le 03.07.2023), *Présentation de l'oeuvre de Jean Piaget* (https://www.fondationjeanpiaget.ch/), (Genève : Fondation Jean Piaget)

Contributeurs Mozilla (28.11.2022), *Concepts de WebAssembly* (https://developer.mozilla.org/fr/docs/WebAssembly/Concepts), (MDN Web Docs: Mozilla)

Malan J. (2022), CS50: Introduction to Computer Science (https://www.youtube.com/watch?v=IDDmrzzB14M), (Harvard University)

Surguy M. (2018), *The History of Processing : Introduction to generative arts and Processing* https://maxoffsky.com/research/research-essay-the-history-of-processing/ (Mark Surguy's blog)

Wikipédia (consulté le 05.07.2023), *Logo (langage)* (https://fr.wikipedia.org/wiki/Logo_(langage) , (San Francisco : Fondation Wikimedia)

Wikipédia (consulté le 06.07.2023), *Histoire des langages de programmation* (https://fr.wikipedia.org/wiki/Histoire_des_langages_de_programmation) , (San Francisco : Fondation Wikimedia)

Wikipédia (consulté le 07.07.2023), *Scratch (programming language)* (https://en.wikipedia.org/wiki/Scratch_(programming_language), (San Francisco : Fondation Wikimedia)

Yegulalp S. (21.03.2023), *How to convert Python to JavaScript (and back again)*, (Needham: InfoWorld), https://www.infoworld.com/article/3209651/how-to-convert-python-to-javascript-and-back-again.html

Illustrations et images

Tortue Logo (illustration 2.1), site web du Projet IDIS (Recherche en Design Image et Son), https://proyectoidis.org/seymour-papert/ (04.07.2023)

Turtle animation (illustration 2.2), Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Turtle-animation.gif (04.07.2023)

Coding with Processing (figure 2.3), Stem Minds (05.07.2023)

Substrate (illustration 2.3), Jared Tarbell, http://www.complexification.net/gallery/machines/substrate/index.php (06.07.2023)

Scène de jeu (illustration 2.4), Sauvez Julie, https://apps.thike.ch/savejulie (06.07.2023)

Block device Icon (figure 2.6), ArchiveIcon, https://www.iconarchive.com/show/tulliana-2-icons-by-umut-pulat/block-device-icon.html (07.07.2023)

programming, window icon (figure 2.6), ArchiveIcon, https://www.veryicon.com/icons/hardware/iconpack-123123/8-programming-windo.html (07.07.2023)

Tower crane (figure 2.6), VeryIcon, https://icons.veryicon.com/png/o/hardware/iconpack-123123/8-programming-windo.png (07.07.2023)

Programming (figure 2.6), VeryIcon, https://icons.veryicon.com/png/System/Soft/Programming.png (07.07.2023)

Action edit icon (figure 2.19), ArchiveIcon, https://www.iconarchive.com/show/crystal-clear-icons-by-everaldo/Action-edit-icon.html (13.07.2023)

Blockly logo (figures 2.14b, 2.19, 2.20), Wikimedia Commons, https://commons.wikimedia.org/wiki/File:WebAssembly_Logo.svg (14.07.2023)

WebAssembly logo (figure 2.20), Google, https://developers.google.com/blockly/guides/app-integration/attribution (14.07.2023)