Yoones MIRHOSSEINI
Marc VINCENT
Sorbonne Université
Master Androïde 2018-19

Rapport du projet IA du module EVIJV : Génération procédurale de niveaux pour *Angry Birds*

Introduction

Ce projet se base sur l'environnement développé dans le cadre de la *3rd Angry Birds Level Generation Competition* organisée à la CIG 2018 (*Conference on Computational Intelligence and Games*) par l'IEEE - *Computational Intelligence Society.* Cette compétition est organisée autour du jeu *Science Birds*, une réimplémentation du célèbre jeu pour mobile *Angry Birds*, développé par la société finlandaise Rovio Entertainment. Le but général de ce projet est de développer un programme capable de générer automatiquement pour ce jeu des niveaux aussi amusants que possible.

Angry Birds est un jeu de puzzle 2D pour mobile dit *physics-based*, c'est-à-dire que la résolution des niveaux y passe par la maîtrise et l'exploitation de la physique du jeu (gravité, résistance des matériaux, etc.). En l'occurrence, le principe de ce jeu est d'utiliser une sorte de lance-pierre pour tirer des oiseaux sur des structures composées de blocs mobiles qui protègent des cochons. Le but du jeu est de détruire ou de perturber ces structures pour exposer les cochons puis les tuer, le tout au moyen des oiseaux. Il existe plusieurs types d'oiseaux et de blocs, dotés de caractéristiques différentes et, dans le cas des blocs, de formes différentes. Le nombre d'oiseaux disponibles varie selon les niveaux et influence le niveau de difficulté. Des explosifs peuvent également être présents dans les niveaux : ils se déclenchent lorsqu'ils touchés.

Dans le cadre des jeux vidéos, la génération procédurale de contenu consiste à créer automatiquement certains éléments d'un jeu, comme des niveaux, des cartes ou n'importe quelle autre structure spécifique à un jeu donné. Son intérêt réside dans le fait qu'elle permet de créer de grandes quantités de contenu, ce qui peut raccourcir le cycle de développement d'un jeu et augmenter sa durée de vie. Elle pourrait aussi être utilisée comme support par les designers de jeux, qui auraient simplement à raffiner ou organiser le contenu généré, plutôt que de le créer entièrement à partir de rien.

Dans le cadre de la compétition, nous avions à notre disposition :

- un clone du *Angry Birds* original, nommé *Science Birds*, implémenté avec Unity3D, qui nous permettra de tester les niveaux générés ;
- un algorithme de génération de niveaux servant d'exemple, écrit en Python. Ce générateur, surnommé *baseline generator*, a été créé en 2016 par Matthew Stephenson, l'un des organisateurs de la compétition, qui en a également décrit le fonctionnement dans un article. Le *baseline generator* crée des niveaux complets comportant des structures où sont placés TNT et cochons, et détermine un nombre d'oiseaux adéquats.

Pour cette compétition, les algorithmes de génération doivent produire un fichier XML doté d'une structure spécifique, qui encode un niveau ; il serait donc possible d'écrire l'algorithme de génération dans le langage de notre choix, du moment qu'il renvoie un fichier adéquat. Néanmoins, étant donné que nous nous basons sur le *baseline generator*, nous utilisons également Python.

Objectif général

Pour avoir une première idée du type de programme que nous devions générer, nous avons analysé le *baseline generator*. Notre attention s'est immédiatement portée sur le fait que les structures produites par ce générateur étaient extrêmement symétriques et, à notre goût, monotones : elles étaient construites comme un empilement de lignes de blocs, chaque ligne ne contenant qu'un seul type de bloc.

Par conséquent, nous nous sommes fixé comme objectif de générer des structures plus variées et asymétriques car nous estimons que les variations de forme des structures contribuent à amuser le joueur et à le lasser moins vite.

Dans ce projet, nous ne nous sommes donc pas occupés du choix du nombre d'oiseaux, de cochons et de TNT et de l'emplacement de ceux-ci : nous avons simplement conservé les algorithmes du *baseline generator* pour ces aspects, et nous avons modifié uniquement la partie du code servant à générer les structures.

La génération de structure dans le *baseline generator*

Afin de mieux mettre en avant les améliorations que nous estimons avoir apporté au *baseline generator*, nous commençons par présenter le fonctionnement de celui-ci.

Pour commencer, le *baseline generator* définit un nombre de structures à créer pour un niveau ; pour chaque structure à créer il définit une zone rectangulaire dans laquelle la structure devra être contenue (donc une hauteur et une largeur maximales). Comme nous l'avons mentionné plus haut, la création d'une structure s'effectue en ajoutant récursivement des lignes de blocs au-dessous de la structure déjà générée, jusqu'à ce que l'on dépasse les dimensions de la zone fixées auparavant. Dans un premier temps, on ne s'intéresse qu'à la disposition des blocs entre eux : l'emplacement réel des blocs dans le monde (notamment leur hauteur) est déterminé uniquement une fois que l'on a fini d'ajouter des lignes.

Une structure est d'abord initialisée avec un certain nombre de pics, tous de même type, qui forment la première ligne de la structure.

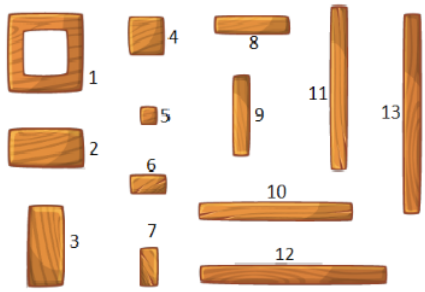


Fig. 2: The thirteen different block types available.

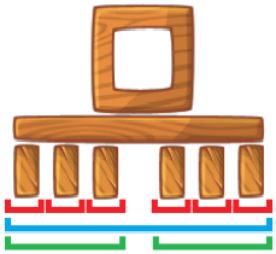


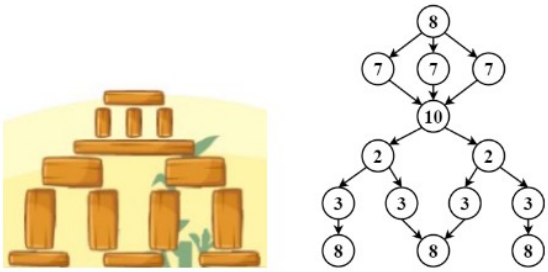
Fig. 3: The bottom row of this structure has three possible subset combinations: each block is in a separate set (red), all blocks are in a single set (blue), and the three left/right blocks are partitioned into two sets (green).

L'ajout d'une ligne s'effectue de la manière suivante. D'abord, un type de bloc parmi les 13 possibles est choisi selon une distribution de probabilités : ce sera le seul utilisé pour toute la ligne. Puis l'algorithme calcule toutes les différentes combinaisons de sous-ensembles pour les blocs de la dernière ligne de la structure actuelle (voir figures ci-dessus). Pour chaque combinaison de sous-ensembles, il y a alors trois possibilités de placement des blocs supports de la ligne à créer :

- placer un bloc sous le milieu de chaque sous-ensemble ;
- placer un bloc sous les deux bords de chaque sous-ensemble ;
- placer des blocs sous le milieu et les deux bords de chaque sous-ensemble.

(a) (b) (c)

Fig. 4: The three possible supporting block placement options for a single block subset: middle (a), edges (b), both middle and edges (c).



(a) (b)

Fig. 5: An example of a generated structure (a) and its corresponding directed acyclic graph representation (b).

Chacune de ces possibilités est essayée pour chaque combinaison de sous-ensembles. On retient que les possibilités valides, c'est-à-dire celles où il n'y a pas de chevauchement entre deux blocs et où chaque bloc de la ligne précédente est supporté par les blocs de la potentielle nouvelle ligne. Ensuite, on sélectionne une possibilité valide au hasard ; si l'on n'en trouve pas, on réitère le processus avec un autre type de bloc. La possibilité sélectionnée est utilisée pour créer la nouvelle ligne.

Une fois la structure créée, les cochons et la TNT sont placés selon des modalités auxquelles nous ne nous intéresserons pas ici.

Réalisations pour notre générateur

Le rendu final du projet consiste en un unique script Python générant des fichiers de niveaux au format défini dans le cadre de la compétition. Il est utilisable dans l'environnement fourni sur le site de la compétition. Notre contribution s'étend jusqu'à la ligne 650, la suite étant présente telle quelle dans le *baseline generator*.

À la suite de notre analyse du *baseline generator*, nous avons largement redéfini, restreint et raffiné les objectifs que nous nous étions fixé dans notre cahier des charges. Ainsi, pour obtenir des structures variées et asymétriques, nous avons rempli successivement plusieurs objectifs précis :

1. **asymétrie au niveau des lignes** : dans le *baseline generator*, une combinaison de sous-ensembles était toujours parfaitement symétrique, et la disposition des blocs supports était la même pour chaque sous-ensemble. Pour améliorer cela, nous avons conservé l'idée d'ajouter récursivement des lignes (fonction *make_structure*), mais nous avons entièrement redéfini la procédure de placement des blocs en abandonnant le principe des sous-ensembles et en tentant de créer des blocs supports pour chaque bloc de la ligne précédente. Pour cela nous avons dû définir des ensembles de conditions complexes pour déterminer les

emplacements valides ; en particulier, nous avons ajouté une option permettant de favoriser le fait qu'un bloc en supporte plusieurs (ce que nous appellerons le *pontage*).

2. **variation des blocs au niveau des lignes** : nous avons fait en sorte qu'une ligne utilise des blocs différents de même hauteur. Nous avons fait un cas particulier pour les pics de la structure, qui peuvent être de n'importe quel type (fonction *make_peaks*).

3. **hauteur des structures** : pour favoriser les structures plus hautes et éviter les structures « plates » (composées d'une seule ligne), nous avons mis en place un mécanisme permettant, lorsque la nouvelle ligne que l'on a créé est trop large, d'essayer d'en créer une autre, avec un nombre limité d'essais.

4. **inversions de blocs entre les lignes** : le *baseline generator* produit uniquement des structures constituées de lignes superposées. Pour faire en sorte d'introduire de la variation dans cette structure très stratifiée, nous avons implémenté un mécanisme d'inversion verticale des blocs : les structures obtenues sont par conséquent nettement plus variées et complexes et ne semblent pas *de visu* avoir été créées ligne par ligne. Ces inversions permettent aussi incidemment d'élaguer des pics

Au passage, nous avons renoncé à utiliser la TNT dans nos niveaux car les algorithmes de placement originaux du *baseline generator* étaient inadaptés à ces nouvelles structures, ce qui donnait des niveaux instables.

Parmi les quatre points listés plus haut, le 2 et 3 ne représentaient pas de difficulté particulière, même s'ils modifient substantiellement l'apparence des niveaux. En revanche les points 1 et 4 ont nécessité un travail de réflexion poussé et seront par conséquent détaillés dans ce qui suit.

Création de lignes asymétriques et variées

Par la suite, nous appelons *dernière ligne* la ligne se trouvant en bas de la structure générée jusque là, et *nouvelle ligne* la ligne à créer sous la dernière ligne.

La nouvelle procédure d'ajout d'une ligne à la structure (*add_new_row*) commence par le choix d'une hauteur constante sur la nouvelle ligne et d'un bloc de départ dans la dernière ligne. Ensuite, nous considérons un par un les blocs de la dernière ligne : pour chacun, nous ajoutons un ou deux blocs sur la nouvelle ligne pour le supporter, selon des modalités détaillées plus loin. L'ordre dans lequel on supporte les blocs de la dernière ligne est le suivant :

- le bloc de départ (*initialize_line*) ;
- les blocs à gauche du bloc de départ, en commençant par le plus à droite (*add_side_blocks*) ;
- les blocs à droite du bloc de départ, en commençant par le plus à gauche (idem).

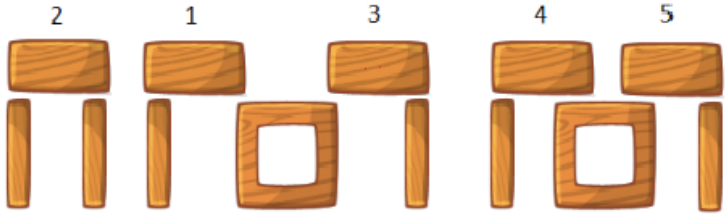


Illustration 1: Exemple d'ordre de traitement des blocs de la dernière ligne

Traiter les blocs dans cet ordre permet de gérer plus facilement les chevauchements. Par rapport au fait de traiter les blocs simplement de gauche à droite, cela permet aussi d'éviter que les lignes dépassent uniquement sur la droite si l'on active l'option de pontage expliquée plus loin.

Lorsque l'on doit supporter un bloc, on vérifie tout d'abord s'il n'est pas déjà supporté entièrement, auquel cas il n'y a rien à faire. Puis on vérifie s'il n'est pas déjà supporté sous un bord (différent selon la direction dans laquelle on est en train d'ajouter les blocs), auquel cas on ajoute un bloc uniquement sous l'autre bord. Sinon, on a le choix entre ajouter un bloc sous chaque bord et ajouter un bloc sous le milieu (c'est-à-dire un bloc qui supporte entièrement le bloc au-dessus). Le choix du type de bloc à ajouter se fait selon une distribution de probabilité sur les blocs de même hauteur que la nouvelle ligne. Ce choix est fait à chaque fois qu'on ajoute un bloc.

Illustration 2: À gauche, le bloc ajouté sous le carré de gauche permet de supporter entièrement le bloc central et le bord gauche du carré de droite. À droite, les deux possibilités de support, en l'absence de support préalable.

Il existe des situations où l'on est obligé de placer un unique bloc support au milieu (*center_mandatory*) : notamment lorsque, étant donné le type de bloc à placer et la nécessité d'éviter le chevauchement, on est contraint de supporter l'ensemble du bloc de la dernière ligne. Inversement, on est obligé de placer des blocs sous les bords si le type de bloc choisi n'est pas assez large pour supporter entièrement le bloc de la dernière ligne (*edges_mandatory*).

Illustration 3: À gauche, à cause du bloc de gauche, le bloc de droite ne peut pas couvrir uniquement un bord du petit carré, mais doit couvrir le carré entier. À droite, le bloc inférieur n'est pas assez large pour supporter à lui seul le bloc supérieur.

Une fois que l'on a décidé de placer un certain type de bloc sous un lieu précis (bord gauche, bord droit, milieu), il faut encore déterminer la coordonnée horizontale exacte du nouveau bloc (*add_block*) (le format des fichiers XML encodant les niveaux requiert les coordonnées du centre du bloc). Pour cela, on détermine l'intervalle dans lequel on peut placer le nouveau bloc (*limits*). Pour ce faire, on raisonne sur l'emplacement des bords des blocs : on vérifie que le nouveau bloc supporte bien le bloc de la dernière ligne aux endroits voulus et qu'il n'y a pas de chevauchement entre le nouveau bloc et le bloc précédent sur la nouvelle ligne. On introduit également trois marges :

- la marge de support, qui définit la « superficie » d'un bloc supérieur que doit couvrir un bloc inférieur pour supporter un bord ;
- la marge de chevauchement, qui est la distance à respecter entre deux blocs du même niveau ;
- la marge de bord, qui représente la distance maximale dont un bloc peut avancer sous un autre tout en supportant son bord.

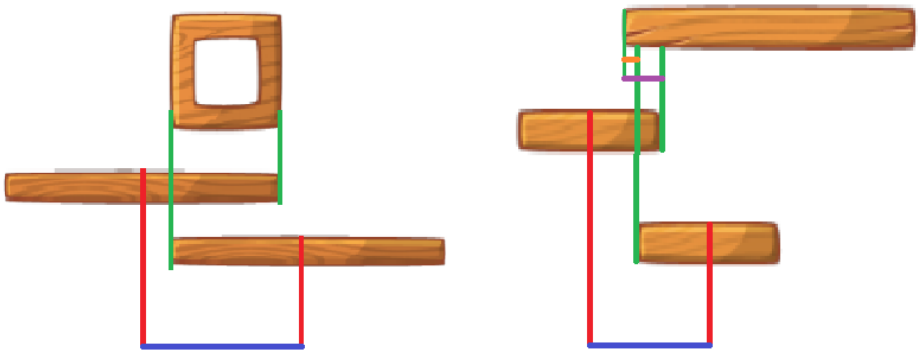


Illustration 4: Exemples d'intervalles de placement possibles : la coordonnée X du centre du bloc support doit se trouver dans l'intervalle bleu. À droite, on voit en orange la marge de bord et en violet la marge de support.

On a également ajouté une option de pontage : une fois que l'on a calculé l'intervalle dans lequel on peut placer le bloc, on regarde si en le plaçant à une extrémité de l'intervalle il pourrait supporter d'autres blocs. Si c'est le cas, avec une probabilité fixée, on le place à cette extrémité. Cette procédure permet de créer des structures beaucoup plus connectées en favorisant le fait de placer un bloc de manière à ce qu'il en supporte plusieurs. La probabilité permet de définir à l'avance le niveau de connexion voulu.

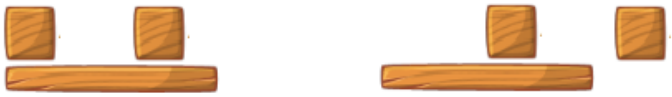


Illustration 5: Lorsque c'est possible, le pontage favorise des structures comme celle de gauche plutôt que comme celle de droite.

Si l'option de pontage ne donne rien, on choisit simplement de manière uniforme aléatoire la coordonnée X du bloc dans l'intervalle.

Inversions de blocs entre les lignes

Une fois la structure créée en entier ligne par ligne, on la parcourt à nouveau, de haut en bas, pour effectuer des inversions verticales de blocs. Cette technique augmente très considérablement la variété des structures et cache le fait que les structures aient été générées ligne par ligne : on voit par exemple apparaître des portions de structure « en escalier ».

Nous avons implémenté deux types d'inversions : les inversions « de U vers Pi » et celles « de Pi vers U », que nous avons surnommé ainsi en raison de la forme des ensembles de blocs à inverser, lorsqu'il y a un seul bloc de chaque côté. Le principe de ces inversions est expliqué dans l'illustration ci-dessous : il s'agit, lorsque certaines conditions détaillées plus loin sont réunies, de décaler un bloc à la ligne supérieure ou inférieure en complétant la ligne d'origine avec des blocs servant à maintenir le support de tous les blocs de la structure. Les inversions U-Pi sont par nature nettement plus fréquentes que celles en Pi-U. Ces deux types d'inversions ont chacune une probabilité fixée d'être effectuées lorsqu'elles sont possibles : on peut ainsi gérer le taux d'inversion voulu pour les niveaux à générer.

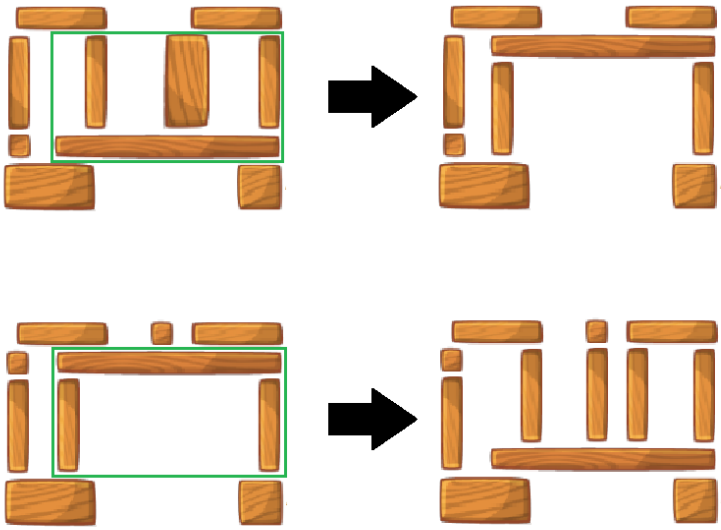


Illustration 6: En haut, une inversion U-Pi. En bas, une inversion Pi-U.

Pour détecter si une inversion est possible, on considère un bloc, dont on détermine les fils (on appelle fils d'un bloc B les blocs qui supportent B et pères de B les blocs qui sont supportés par B). S'il n'a qu'un fils, on vérifie que les pères de ce fils n'ont pas d'autres fils et qu'ils sont tous de même hauteur : ce sont les conditions nécessaires pour réaliser l'inversion d'une forme en U à une forme en Pi.

S'il a plusieurs fils, on vérifie que ces fils n'ont pas d'autres pères et que le bloc considéré serait supporté s'il était descendu : ce sont les conditions nécessaires pour réaliser l'inversion d'une forme en Pi à une forme en U. Pour réaliser cette inversion, on détermine tous les emplacements auxquels il faudra placer des blocs pour soutenir les blocs qui étaient auparavant soutenus par le bloc que l'on va descendre.

Un effet inattendu des inversions Pi-U a été de produire une sorte d'élagage des pics. En effet, lorsqu'un pic est décalé à la ligne du dessous, il n'y a aucun bloc à ajouter à la première ligne puisqu'il n'y a aucun bloc à supporter au-dessus : dans les faits, on enlève ainsi un pic (ou plutôt, on le décale à la ligne du dessous). Nous avons introduit une restriction pour éviter ce type d'élagage s'il ne reste qu'un pic, afin de conserver la hauteur de la structure.

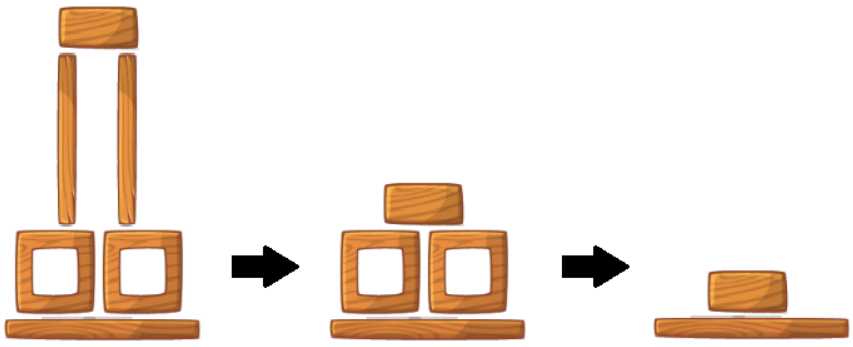
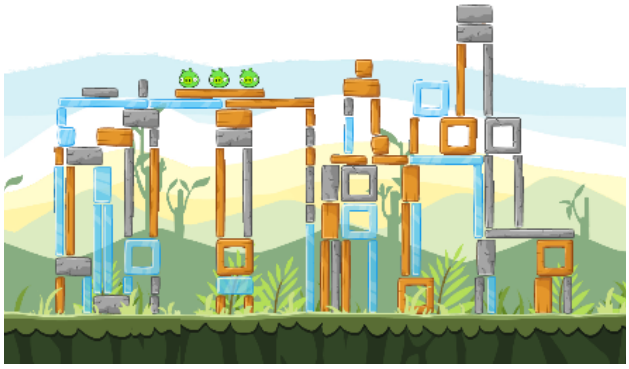
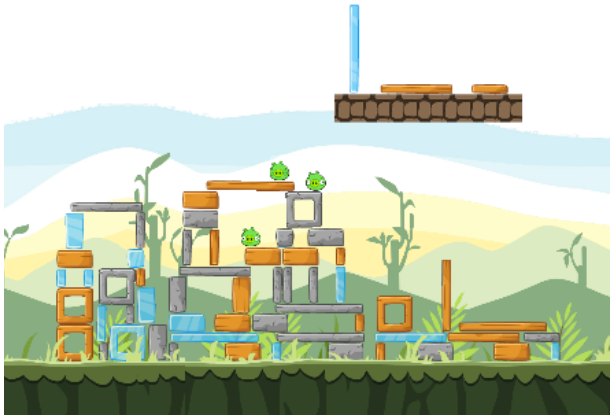
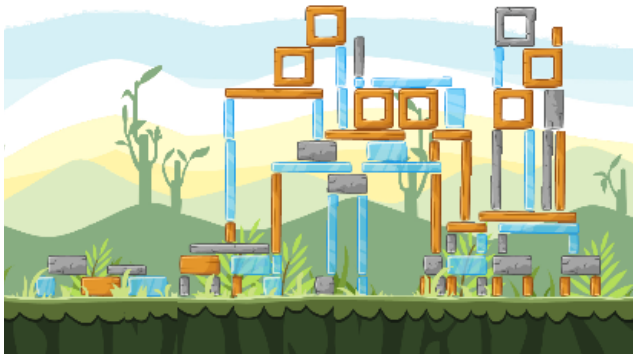


Illustration 7: Exemple de processus de "raccourcissement" d'un pic

Exemples de structures obtenues



Bugs connus

Nous n'avons pas eu le temps de régler tous les bugs que nous avons rencontré, par conséquent le générateur peut une fois de temps en temps produire des niveaux instables. Dans le détail : le fait de faire de nombreuses inversions dans de grandes structures peut déstabiliser celles-ci et il arrive qu'un cochon tombe d'une plate-forme dès le lancement du niveau.

Conclusion

Notre approche a permis d'augmenter substantiellement la variété des structures produites en créant de l'asymétrie et en mélangeant les lignes. Parmi les améliorations que nous avions envisagé, nous pouvons citer la mise en place d'un mécanisme de fusion de blocs, où deux blocs superposés de même taille seraient remplacés par un unique bloc deux fois plus grand. Il faudrait par ailleurs remplacer les fonctions de placement de la TNT et des cochons du *baseline generator* par des fonctions plus adaptées aux nouvelles structures.

Références

Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games, Matthew Stephenson, Jochen Renz, CIG 2018