Projet final : particules

Initiation au développement

BUT informatique, première année

Ce projet vise à mettre en place un système de particules. Il s'agit d'un outil fréquemment utilisé dans les jeux vidéos, les logiciels de 3D, les effets spéciaux pour le cinéma, etc. Un tel système permet de simuler graphiquement des effets complexes à représenter : explosions, feu, pluie, et bien d'autres. En général, les systèmes de particules sont utilisés en trois dimensions, cependant, pour simplifier, celui que vous allez développer travaillera en deux dimensions (principalement car l'affichage en trois dimensions serait assez complexe). La figure 1 montre deux exemples d'images que vous pourrez engendrer à l'aide de votre code (elles ont été produites à partir de ma version du projet).

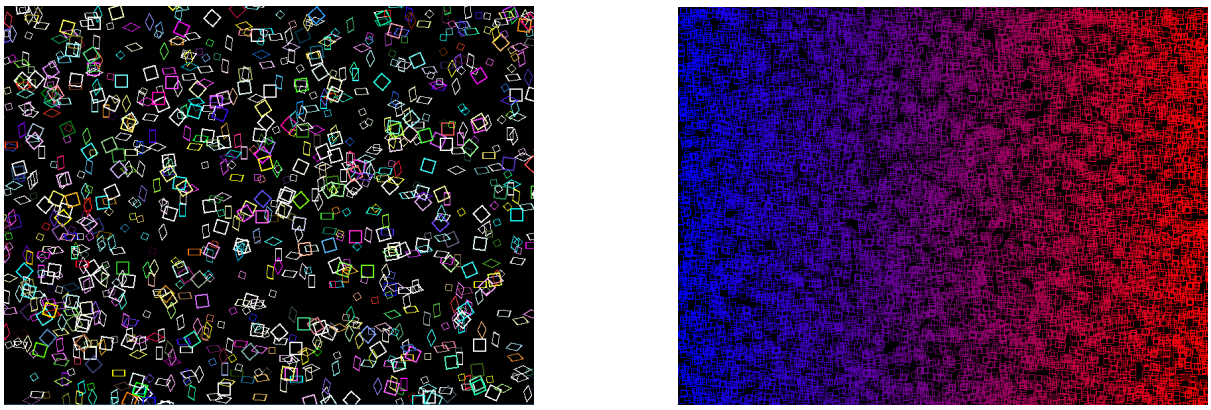

FIGURE 1 – Deux images engendrées par un système de particules en deux dimensions. À gauche, 1000 particules engendrées aléatoirement (couleur, position, taille en abscisse, taille en ordonnée, rotation). À droite, 10000 particules engendrées un peu moins aléatoirement : la couleur dépend de la position sur l'axe des abscisses.

1 Évaluation et dates

Ce projet comptera à la fois pour le cours d'introduction au développement et pour la SAÉ implémentation d'un besoin client.

1.1 Dates

— C'est le travail de la partie 4 ci-dessous qui comptera pour le cours d'introduction au développement. Vous devrez le rendre au plus tard le **dimanche 9 janvier à 23h55**.
— C'est le travail de la partie 5 ci-dessous qui comptera pour la SAÉ implémentation d'un besoin client. Vous devrez le rendre au plus tard le **dimanche 23 janvier à 23h55**.

1.2 Critères d'évaluation

Pour les deux parties, seront évalués (par ordre d'importance) : le bon fonctionnement du code, le choix et la qualité des cas de tests, la qualité du code (choix judicieux de noms de variables et fonctions,

organisation claire des fichiers, documentation bien ciblée, etc), l'efficacité du code (capacité à gérer beaucoup de particules en même temps).

Pour la SAÉ, le nombre d'extensions réalisées sera aussi un critère important d'évaluation (il ne s'agit pas de réaliser toutes les extensions, bien sûr, mais d'en réaliser plusieurs et pas seulement les plus simples).

1.3 Rendre son travail

Pour rendre votre travail vous utiliserez obligatoirement un dépôt sur le Gitlab de l'Université (accessible à l'adresse `https://gitlab.univ-nantes.fr/`) à l'aide du quel vous travaillerez tout au long du projet.

Au moment de chaque rendu vous créerez une *release* sur votre dépôt (dans le menu de gauche d'un dépôt, faire *Deployments* puis *Releases* et suivre les instructions). C'est le contenu de cette *release* qui sera évalué.

2 Principe général d'un système de particules

Un système de particules gère en ensemble de particules en mettant à jour leurs caractéristiques de manière régulière au cours du temps.

2.1 Particule

Une particule est un objet possédant :

— un certain nombre de caractéristiques permettant de l'afficher à l'écran : au minimum une position, mais éventuellement aussi un angle de rotation, une taille (ou une échelle par rapport à la taille d'une particule de base), une couleur, une transparence, etc.

— un certain nombre de caractéristiques permettant de faire évoluer son état au cours du temps : vitesse, accélération, durée de vie, fonctions d'évolution de la couleur ou de la transparence, capacité d'interaction avec les autres particules, etc.

2.2 Système de particules

Un système de particules gère un ensemble de particules en :

— engendrant de nouvelles particules au fil du temps en fonction de ses caractéristiques (taux de génération, position du générateur, etc),

— mettant à jour ses particules (en particulier leur position) au fil du temps en fonction des caractéristiques de celles-ci (en particulier leur vitesse),

— supprimant éventuellement les particules dont la durée de vie est épuisée.

3 Organisation des sources du projet

Des sources vous sont fournies avec le projet pour vous aider à implanter plus facilement votre système de particules. Vous devez respecter leur organisation tout au long du projet et vous ne devez pas la changer (si vous avez vraiment besoin de le faire ça doit être avec l'accord d'un enseignant et il faudra pouvoir le justifier).

3.1 Dossier principal

Le dossier principal du projet contient les fichiers go.mod et go.sum, le code de la fonction main (et des fonctions qu'elle utilise), un fichier config.json qui sert à configurer le projet, et les autres dossiers du projet.

Fichiers go. Les fichiers de code go qu'on trouve dans ce dossier ne doivent pas être modifiés (à part éventuellement pour certaines extensions, quand c'est précisé). Ils se chargent de mettre à jour et d'afficher votre système de particules à un rythme régulier, soixante fois par seconde. Pour cela ils utilisent une bibliothèque tierce : `github.com/hajimehoshi/ebiten/v2`.

Le programme peut fonctionner dans deux modes :

mode debug : le nombre moyen d'images par seconde est affiché à l'écran, celui-ci devrait être d'environ 60, si ce n'est pas le cas c'est que votre code n'est pas assez efficace (ce mode est activé quand le champ Debug du fichier config.json vaut $true$),

mode normal : le nombre moyen d'images par seconde n'est pas affiché (ce mode est activé quand le champ Debug du fichier config.json vaut $false$ – ou si ce champ n'existe pas).

> Durant tout le projet, si vous ajoutez des informations de debug (par exemple avec la bibliothèque log), laissez-les dans votre code en les mettant dans une conditionnelle qui permettra de les afficher seulement en mode debug (vous pouvez voir un exemple de comment faire cela dans le fichier Draw.go). Ceci permettra à la personne qui corrigera votre projet de voir plus en détails ce qui se passe si elle le souhaite, sans avoir à ajouter elle-même du code Go.

Fichier de configuration. Le fichier config.json permet de configurer le projet. Pour le moment il permet de définir la taille de la fenêtre d'affichage et son titre, de définir l'image utilisée pour représenter les particules, et de définir si on est en mode debug ou pas. Quelques autres paramètres sont aussi déjà disponibles dans ce fichier, qui seront utiles pour la première partie du projet mais qui n'ont pas d'effets particuliers pour le moment tant que vous n'avez pas écrit votre code.

3.2 Dossier assets

Le dossier *assets* contient les images utilisées pour le projet (une seule image en fait, le carré blanc représentant une particule) et les fonctions qui permettent de les charger en mémoire.

Vous n'avez normalement jamais à toucher au contenu de ce dossier.

3.3 Dossier config

Le dossier *config* contient le code permettant de lire le fichier de configuration config.json.

Vous pouvez éventuellement ajouter des choses ici si vous voulez enrichir ce fichier de configuration (ce qui peut permettre de tester différents paramètres de votre système de particules sans avoir à recompiler le programme à chaque fois : nombre de particules, point d'origine, gravité, etc).

3.4 Dossier particles

Le dossier *particles* contient le squelette du code que vous avez à écrire pour votre projet. Vous pouvez y ajouter autant de fichiers que vous voulez et vous pouvez ajouter du code dans les fichiers qui existent déjà. Cependant, il ne faut pas modifier les prototypes des fonctions existantes ni enlever des champs des types déjà définis (vous pouvez en ajouter par contre) car ils sont utilisés par les fonctions du main.

Fichier type.go. Ce fichier définit les types de bases pour un système de particules : le système lui même (contenant un tableau de particules) et la particule (avec une position, une rotation, une échelle, une couleur et une transparence).

Vous serez très probablement amenés à enrichir ces types.

Fichier new.go Ce fichier donne le prototype d'une fonction NewSystem() qui initialise un système de particules.

Vous devrez écrire cette fonction.

Fichier update.go Ce fichier donne le prototype d'une fonction Update() qui met à jour un système de particules. Cette fonction est appelée 60 fois par seconde par le main du programme principal sur un système de particules initialement créé par NewSystem().

Vous devrez écrire cette fonction. Dans toute la suite, quand on vous demande de modifier une fonction Update() c'est de celle-ci qu'il s'agit, à moins qu'on précise autre chose.

4 Premier travail : système de particules de base

Votre premier travail sera de coder un système de particules basique : chaque particule a une vitesse initiale (qui n'évolue pas au cours du temps), une position initiale, et sa position évolue au fil du temps en fonction de sa vitesse. Les autres caractéristiques des particules (taille, rotation, couleur, transparence) n'évoluent pas.

4.1 Fonction NewSystem()

Dans un premier temps il faudra coder la fonction NewSystem() (fichier particles/new.go) afin qu'elle crée un certain nombre de particules en leur donnant à chacune une position initiale.

Les contraintes suivantes devront être respectées :

— le nombre de particules devra être InitNumParticles, qui est défini dans le fichier config.json (il faudra donc l'obtenir en lisant la variable General du paquet config),

— la position des particules devra être soit aléatoire soit la même pour toutes les particules, en fonction du champs RandomSpawn du fichier config.json (encore une fois, il faudra utiliser la variable General du paquet config pour obtenir cette valeur),

— si la position des particules est aléatoire il faudra utiliser le paquet math/rand pour la définir pour chaque particule, en s'assurant qu'elle reste toujours à l'intérieur de l'écran (la taille de l'écran se trouve dans la variable General du paquet config),

— si la position des particules n'est pas aléatoire elle devra être définie par les champs SpawnX et SpawnY du fichier config.json (encore une fois, il faudra utiliser la variable General du paquet config pour obtenir ces valeurs).

Attention, les coordonnées à l'écran sont données en pixels à partir du coin supérieur gauche de la fenêtre : quand x augmente on se déplace vers la droite, quand y augmente on se déplace vers le bas.

Attention, par défaut, une particule a une taille de 0, est transparente et n'a pas de couleur, elle n'est donc pas visible. Si vous voulez rendre vos particules visibles il faut changer ces trois caractéristiques.

— La taille est définie par les champs ScaleX et ScaleY, s'ils valent tous les deux 1 la particule a sa taille de base (celle de l'image particle.png du dossier assets). Faire varier ScaleX (respectivement, ScaleY) modifie la taille de la particule selon l'axe des abscisses (respectivement, des ordonnées). La valeur indiqué dans ces champs multiplie la taille (avec 0.5 on divise la taille par deux, avec 2 on la multiplie par 2).

— La transparence est définie par le champ Opacity. S'il vaut 1 la particule est complètement opaque et s'il vaut 0 elle est complètement transparente. Toutes les valeurs entre ces deux extrêmes sont possibles.

— La couleur est définie par les champs ColorRed, ColorGreen et ColorBlue. Pour chacun d'entre eux, une valeur de 1 indique la présence de cette couleur à 100% et une valeur de 0 indique l'absence de cette couleur. Ainsi, si ces trois champs valent 0, la particule sera noire, et s'ils valent tous les trois 1, la particule sera blanche.

La fonction NewSystem() qui vous est fournie initialement affiche une particule au centre de l'écran.

4.2 Fonction Update() : vitesse des particules

Il faudra ensuite enrichir la fonction NewSystem() pour donner une vitesse initiale aux particules et coder la fonction Update() (fichier particles/update.go) pour les mettre à jour en fonction de cette vitesse.

Attention, c'est la fonction Update() du fichier update.go situé dans le répertoire particles qu'il faut modifier et surtout pas celle du fichier update.go situé à la racine du projet.

— Les vitesses seront définies selon deux coordonnées (x et y), on aura donc une vitesse pour chaque coordonnée,
— chaque particule aura ses propres vitesses, définies aléatoirement (utiliser math/rand) et une fois pour toutes par NewSystem(),
— la fonction Update() se contentera, pour chaque particule d'ajouter sa vitesse selon la coordonnée x à sa position en x et sa vitesse selon la coordonnée y à sa position en y,
— la fonction Update() est automatiquement appelée 60 fois par seconde, de manière régulière, par le programme principal (déjà codé) du projet.

4.3 Fonction Update() : génération de particules

Enfin, il faudra enrichir votre fonction Update() pour engendrer de nouvelles particules au cours du temps. Pour cela, on dispose d'un taux de génération (SpawnRate dans la config) qui indique combien de nouvelles particules doivent être engendrées à chaque appel à la fonction Update(). Il s'agit d'un nombre décimal (ainsi, s'il vaut 3 on doit engendrer 3 particules lors de chaque appel à Update() et s'il vaut 0.5 on doit engendrer une nouvelle particule lors d'un appel à Update() sur deux).

— Les nouvelles particules seront engendrées comme les particules de départ (position aléatoire ou la même pour toutes, selon la configuration, vitesse aléatoire),
— et elles seront bien sûr mises-à-jour comme les autres par Update() pour faire évoluer leur position au cours du temps.

Quand vous choisissez aléatoirement les vitesses des particules, vous pouvez plus ou moins contraindre leur choix. Ceci changera l'aspect de votre système de particules (par exemple des vitesses complètement aléatoires en x et y, aussi bien négatives que positives, avec toutes les particules engendrées au même point donnent un effet d'explosion). Faites des essais variés. Vous pouvez par exemple avoir plusieurs fonctions de génération des vitesses dans votre code, et permettre de sélectionner celle qu'on veut en ajoutant un champs pour cela au fichier de configuration.

5 Deuxième travail : extensions

La première partie du projet était très cadrée, cette deuxième partie est beaucoup plus libre. Vous devez coder quelques unes des extensions ci-dessous (elles sont à peu près classées par difficulté, de la plus simple à la plus compliquée, n'en faites pas que des simples, n'hésitez pas à interagir avec vos professeurs pour choisir vos extensions). Vous pouvez aussi proposer de nouvelles extensions, mais celles-ci doivent être validées par un professeur avant que vous commenciez à travailler dessus.

Chaque extension doit pouvoir être activée et désactivée en utilisant le fichier de configuration config.json (vous pouvez vous inspirer du fonctionnement du mode debug, activable dans config.json, pour cela). De même, si elles s'y prêtent, vos extensions doivent pouvoir être paramétrées dans le fichier config.json.

Vous rendrez votre travail avec un fichier de configuration config.json construit pour qu'en lançant votre projet sans rien toucher on obtienne un résultat à l'écran qui soit le plus impressionnant/joli possible. Vous pouvez aussi fournir d'autres fichiers de configuration (appelez-les config2.json, config3.json, etc) si vous souhaitez montrer différentes possibilités intéressantes de votre système de particules.

5.1 Gravité

Cette extension consiste à modifier vos particules pour prendre en compte la gravité. Pour cela, on va simplement utiliser une constante (l'accélération de la gravité),qui sera définie au niveau du système de particules, et prendra une valeur paramétrée par config.json. À chaque mise-à-jour d'une particule, cette valeur sera ajoutée à la vitesse selon la cordonnée y (l'ordonnée) de cette particule.

5.2 Extérieur de l'écran

Une particule qui est largement sortie de l'écran, de telle sorte qu'elle ne risque plus d'y revenir (ou en tout cas qu'on ne s'en rendra pas compte si elle ne revient pas), n'a pas de raison de continuer à être mise-à-jour et affichée. Pour détecter cela, on pourra utiliser un nouveau champs dans le fichier de configuration, indiquant la marge en pixels en dehors de l'écran en dehors de laquelle on arrête de mettre-à-jour et d'afficher une particule. Dans la fonction Update() on ne mettra à jour que les particules qui sont encore dans la zone définie par cette marge. De plus, on n'affichera aussi que ces particules. Pour cela, il faut modifier légèrement la fonction Draw() du fichier draw.go : il suffit d'ajouter une condition autour du code à l'intérieur de la boucle for.

5.3 Durée de vie

Il peut être intéressant, dans un système de particules, de donner une durée de vie aux particules. On peut ainsi représenter certains phénomènes plus facilement (par exemple, on peut représenter du feu en ayant des particules bougeant vers le haut et avec une durée de vie assez courte, les faisant disparaître — idéalement en devenant de plus en plus transparentes — au bout d'un certain temps). Cela signifie que, après un certain nombre d'appels à la fonction Update(), une particule arrête d'être mise-à-jour et d'être affichée. Pour mettre en œuvre cela il faudra ajouter un champs dans le fichier de configuration, permettant de définir la durée de vie (en nombre d'appels à Update()) des particules. Chaque particule comptera combien de fois elle a été mise-à-jour et, si elle dépasse la durée de vie, ne sera plus mise-à-jour ni affichée (cette deuxième partie demande de modifier légèrement la fonction Draw() du fichier draw.go, il suffit d'ajouter une condition autour du code à l'intérieur de la boucle for).

5.4 Variations de couleur, d'échelle, de rotation, de transparence

L'objectif d'un système de particules reste en général d'obtenir des résultats intéressants visuellement (même si on peut aussi les utiliser pour modéliser et observer des phénomènes physiques réalistes). Pour cela, il peut être pertinent de faire varier les caractéristiques visuelles des particules au cours du temps : couleur, taille, orientation, transparence.

En pratique, pour éviter de coder cela en dur dans la fonction Update() et pour permettre un peu de modularité, on va associer, au système et/ou à chaque particule des fonctions chargées de faire évoluer ces différentes caractéristiques. Ces fonctions seront appelées sur chaque particule à chaque appel à Update().

Ceci est rendu possible en Go car une fonction est une valeur comme une autre, on peut donc avoir un champ d'un type structuré qui soit une fonction. On a alors la possibilité de définir un certain nombre de fonctions différentes de mise-à-jour et d'associer l'une d'entre-elles (à l'exécution) au système ou à chaque particule, en fonction de valeurs de champs du fichier de configuration.

Pour donner un exemple : on pourrait avoir une fonction increaseOpacity(p *Particle) qui augmenterait un peu l'opacité d'une particule à chaque fois qu'on l'appelle et une fonction decreaseOpacity(p *Particle) qui diminuerait un peu l'opacité d'une particule à chaque fois qu'on l'appelle. Dans le fichier

de configuration on aurait un champ qui permettrait de choisir entre ces deux fonctions (un entier par exemple, 0 correspondant à aucune fonction, 1 à increaseOpacity(p *Particle) et 2 correspondant à decreaseOpacity(p *Particle)). En fonction de cette valeur, on mettrait la fonction de gestion de l'opacité du système de particules soit à func(p *Particle), soit à increaseOpacity soit à decreaseOpacity. Et dans tous les cas, Update() appellerait la fonction de gestion de l'opacité du système sur chaque particule.

Cette partie est très vaste, il ne s'agit pas de tout faire, mais plutôt de proposer quelques fonctions de mise-à-jour des caractéristiques visuelles des particules qui vous paraissent intéressantes.

5.5 Optimisation de la mémoire

Cette partie nécessite d'avoir fait au moins l'extension 5.2 ou l'extension 5.3.

Une particule qui n'est plus mise-à-jour ni affichée (on dira qu'elle est morte) n'a pas besoin d'être gardée en mémoire. Si on laisse le système de particules engendrer de nouvelles particules pendant longtemps, le tableau qui les contient ne cesse jamais de croître et contient de plus en plus de particules mortes. Pour éviter cela, on peut supprimer les particules mortes du tableau ou, plus simplement, les garder à la fin de celui-ci :

— À chaque fois qu'une particule devient morte, on échange sa place avec la dernière particule vivante (non morte) du tableau.
— À chaque fois qu'on engendre une nouvelle particule on la place là où est la première particule morte s'il y en a une (dans le cas contraire, on augmente, comme avant, la taille du tableau pour y mettre la nouvelle particule).

Pour être efficace, ceci demande de mémoriser à tout instant (dans un champs du type System la position de la première particule morte).

5.6 Plus d'optimisation de la mémoire

Cette partie nécessite d'avoir fait au moins l'extension 5.2 ou l'extension 5.3.

La méthode d'optimisation de la mémoire qui est présentée à l'extension 5.5 est valable tant que les particules contiennent relativement peu d'informations. Cependant, dès qu'on déplace une particule dans le tableau, on copie tout son contenu. Ceci peut ralentir le programme s'il y a beaucoup de particules et/ou si celles-ci ont beaucoup de contenu (beaucoup de champs). Pour éviter ces recopie, on peut considérer que le type System contient un tableau de pointeurs vers des particules au lieu d'un tableau de particules.

Cependant, si on a des pointeurs vers des particules, lorsqu'on va remplacer les éléments correspondants dans le tableau (comme suggéré dans l'extension 5.5), les particules existeront toujours en mémoire, c'est simplement leur adresse qui aura été supprimée du tableau. Ceci pourrait finir par remplir tout l'espace disponible. Pour éviter cela, le (*Runtime* du) langage Go met en place un ramasse-miettes (*garbage collector*) qui va, de temps en temps, nettoyer la mémoire pour supprimer tous les éléments dans celle-ci qui ne sont plus référencés (pour lesquels il n'existe plus de variable qui contient leur adresse dans le programme) et donc en particulier nos particules mortes qui ne sont plus référencées dans le tableau. Plus il y a de mémoire à nettoyer et plus cela prend du temps. De plus, pour des raisons de sécurité et d'intégrité des données, le fonctionnement normal du programme doit être interrompu pendant le nettoyage de la mémoire. On peut donc observer des ralentissements si on supprime des particules du tableau.

Pour résoudre ce problème on peut réutiliser les particules mortes : quand on doit ajouter une particule au tableau, au lieu de remplacer une particule morte (comme dans l'extension 5.5) on va réutiliser cette particule en modifiant ces caractéristiques (position, vitesse, etc) pour faire comme si c'était une nouvelle particule.

5.7 Forme du générateur

Pour l'instant, votre système de particules devrait engendrer des particules de deux manières différentes : aléatoirement positionnées à l'écran ou toutes en un même point. Il peut être intéressant de permettre d'engendrer des particules sur des formes plus complexes qu'un point. Par exemple, on pourrait les engendrer aléatoirement réparties sur un cercle.

Dans ce contexte il devient possible de choisir les vitesses initiales des particules en fonction de la forme du générateur. Par exemple, on pourrait faire que les particules sont initialement propulsées vers l'extérieur d'un cercle.

À vous de voir comment mettre ça en œuvre et comment le paramétrer dans le fichier de configuration.

> Cette partie est très vaste, il ne s'agit pas de tout faire, mais plutôt de proposer quelques formes de générateur et quelques choix de vitesses initiales associés qui vous paraissent donner des résultats intéressants.

5.8 Collisions

Les particules dans votre systèmes n'ont pour l'instant aucune interactions entre elles. Il peut être intéressant de détecter quand deux particules se rencontrent (il y a une collision) et de réagir à ceci d'une manière ou d'une autre (en changeant la couleur, la taille, l'orientation, la vitesse, etc des particules qui se sont rencontrées).

On peut même imaginer simuler les interactions physiques entre les particules : lors d'une rencontre elles se repoussent.

> **Attention :** les deux extensions suivantes sont beaucoup plus difficiles que les précédentes.

5.9 Déplacement du générateur

Le générateur de votre système de particules est immobile. Parfois on souhaiterait pouvoir le déplacer (imaginez par exemple qu'on l'utilise pour représenter du feu sur une torche portée par un personnage en mouvement) mais ceci crée des complexités : les particules doivent-elles suivre le générateur ? doivent-elles rester à leur place ? le déplacement du générateur influe-t-il sur la vitesse initiale des particules ?

Par ailleurs, ceci nécessite de modifier (légèrement) la fonction Update() principale du projet (pas celle que vous avez modifiée jusqu'à présent, mais celle qui agit sur un *game). C'est là qu'on va pouvoir gérer l'interaction avec le clavier et/ou la souris pour y réagir et modifier la position du générateur du système de particules. Pour savoir comment faire cela, il faudra consulter la documentation de la bibliothèque Ebiten : `https://pkg.go.dev/github.com/hajimehoshi/ebiten/v2`

5.10 Modification dynamique du système de particules

Même avec le fichier de configuration ce n'est pas toujours simple de trouver les bons paramètres pour avoir un résultat conforme à nos attentes à l'écran. On peut passer par un bon nombre d'essais où on modifie légèrement la configuration puis on teste à nouveau le programme. Il serait intéressant de pouvoir changer les paramètres du système à l'exécution, par exemple en utilisant le clavier pour sélectionner et modifier les différents paramètres.

Ceci demande de développer une petite interface. Il faudra donc modifier la fonction Draw() pour afficher l'interface et la fonction Update() principale du projet (pas celle que vous avez modifiée jusqu'à présent, mais celle qui agit sur un *game) pour gérer l'interaction avec le clavier et/ou la souris. Pour savoir comment faire cela, il faudra consulter la documentation de la bibliothèque Ebiten : `https://pkg.go.dev/github.com/hajimehoshi/ebiten/v2`