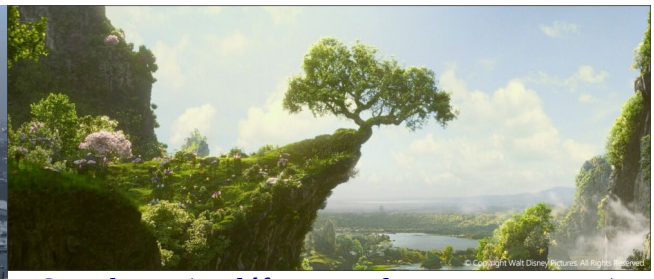
PROJET
Générateur paramétrique de paysages
(*Parametric landscapes generator*)

Domaine

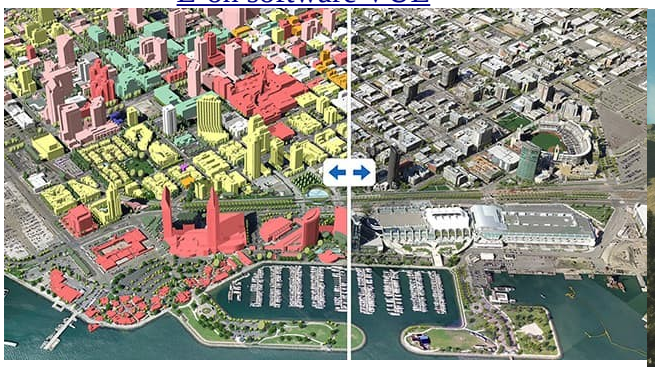
Le domaine d'application abordé par ce projet est celui des contenus créatifs numériques générés, ou « génératifs ». Que ce soit pour des projets d'architecture ou d'urbanisme, pour le cinéma réaliste ou en images de synthèse, et bien sûr pour l'industrie du jeu vidéo, il y a un besoin de contenus virtuels riches et complexes pour « remplir » les mondes virtuels, en général en 3D. Des armées de graphistes et de modeleurs 3D sont là pour donner corps aux visions des créateurs, mais quand on veut des mondes ouverts (jeux « mondes ouverts ») ou qu'on veut donner sur grand écran des visions larges et très détaillées de scènes imaginaires ou de reconstitutions historiques, ou qu'on est une petite structure d'architectes ou d'urbanistes n'ayant pas les moyens des grosses productions, on peut se tourner vers des logiciels qui vont « imaginer » une scène en appliquant des règles pour fournir un contenu d'un niveau de détail sans limites (selon la puissance de la machine) et plus ou moins réalistes (selon le niveau d'expertise injecté dans le générateur).

Les 1ers générateurs de terrains virtuels étaient plus une affaire de passionnés se prenant un peu pour des dieux. Les générateurs de mondes sont maintenant devenus une (petite) industrie avec des entreprises qui se spécialisent, dans les plantes, dans les villes, dans la création de planètes !

E-on software VUE



SpeedTree (Maléfique, Walt Disney Pictures)



Esri CityEngine

World Machine et VUE

Bien sûr tout ceci ne se fait pas tout seul en claquant des doigts : il faut programmer une application qui met en place des algorithmes génératifs qui sculptent des formes et les habille avec des textures, en respectant ou en simulant des principes naturels ou artificiels : érosion des sols, croissance des plantes, architecture... Et il faut doter ces applications d'interfaces pour permettre aux créateurs de guider les processus et obtenir le résultat voulu, sans avoir à entrer tous les détails !

La très grande majorité des modèles 3D actuels utilisent des surfaces polygonales, en fait des maillages triangulés, dont vous devez maintenant commencer à comprendre les bases. Ajoutez un z au (x,y) de vos Coords et vous avez une surface 3D ! Mais ceci est facultatif... ce qui nous intéresse dans ce projet ce n'est pas tant l'aspect 3D que l'aspect **vectoriel** : en effet contrairement au domaine de l'image 2D où la **matrice** de pixels reste très importante pour les logiciels de création (Photoshop...), la 3D doit par nécessité décomposer les formes plutôt que d'enregistrer de manière homogène une grille de points de l'espaces (voxels). Et là où de lourdes matrices de pixels homogènes se traitent assez bien en procédural (en langage C par exemple), la variété d'objets et la complexité de leurs interactions rend très pertinente l'utilisation de langages orientés objets pour le vectoriel. De plus si nous voulons des scènes à la fois vectorielles et riches en détails nous voulons gérer des centaines de milliers d'objets, càd que nous avons besoin d'un langage orienté objet **et** véloce : nous avons besoin du C++ !

Objectifs

Vous allez proposer votre propre générateur de paysages vectoriels ! Il n'est pas demandé d'obtenir un résultat ultra-réaliste en 3D image de synthèse. Il est demandé que votre générateur possède un minimum de style (c'est le votre) et surtout qu'il y ait assez de complexité dans les objets graphiques générés et leurs relations pour que vous puissiez mettre en œuvre les techniques de programmation orientée objet vues en cours. Vous ne devez pas utiliser un « framework » de création de haut niveau (tels que les outils de génération de scènes des liens précédents) puisque justement vous devez apprendre (les bases de) la création de tels logiciels.

Votre générateur de paysage sera un générateur : il sera possible d'obtenir une « infinité » de paysages différents en relançant ses procédures de génération. Ceci implique un usage maîtrisé des générateurs aléatoires. Bien sûr il est très difficile de faire un générateur qui fabrique des paysages complètement différents d'une exécution à l'autre : ce qui est demandé ce sont des **variations** à l'intérieur d'une **enveloppe**.

Votre générateur de paysage sera paramétrique : il sera possible de **guider** les résultats obtenus en indiquant par l'intermédiaire de l'interface (mode console) des valeurs définissant le cadre des tirages aléatoires. Par exemple il sera possible de passer d'une végétation dense (jungle) à une végétation rare (désertique) en réglant un paramètre de densité de végétation. La suite de l'énoncé précisera cet aspect.

Votre générateur de paysage sera vectoriel 2D, il s'appuiera sur le format SVG, format qui a déjà été utilisé en TPs pour les exercices sur les centres de gravité et pour le maillage triangulé. **Les codes fournis à ces occasions sont évidemment (ré)utilisables en tout ou partie dans le cadre du projet.** Aucun autre code ne sera fourni. Il sera possible de réussir très bien le projet uniquement sur la base de ces codes déjà fournis (modules util et Svgfile) mais il pourrait être intéressant de les compléter et d'étudier un peu le format SVG (ceci reste facultatif).

On ne demandera donc pas un aspect 3D complet avec positionnement de caméra, rotations dans l'espace, perspectives et tutti quanti. Ce qui est demandé est un « **effet de profondeur** » qui peut être obtenu, de la façon la plus simple, par une simple *juxtaposition de plans ou de calques : un avant plan cache un plan médian qui cache un arrière plan*. Il sera demandé d'accorder un soin particulier à l'**ordre d'affichage** (ordre des éléments graphiques ajoutés au fichier SVG) qui garantira que les objets qui apparaissent visuellement devant cachent ceux qui apparaissent derrière.

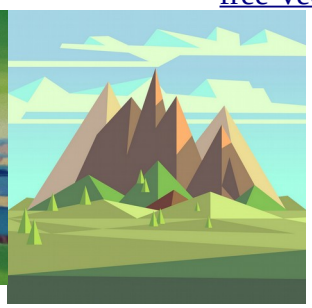
L'univers graphique est totalement libre. Dans le cadre d'un projet étudiant (non commercial) vous pouvez vous inspirer de créations avec copyrights. Page suivante je mets quelques paysages qui peuvent servir **d'inspiration**, **sans aucun caractère obligatoire ni de ressemblance exacte.**

Les images suivantes ont été trouvées avec **Google image** avec les recherches suivantes « landscape vector art » « stylised landscape low poly » « stylised landscape vector » …

https://imgur.com/gallery/NtTBh

free-vector-landscape

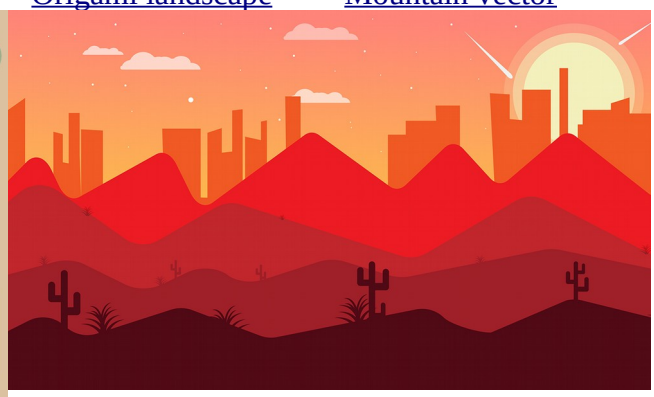
Planète Namek (Univers Dragon Ball)



Origami landscape

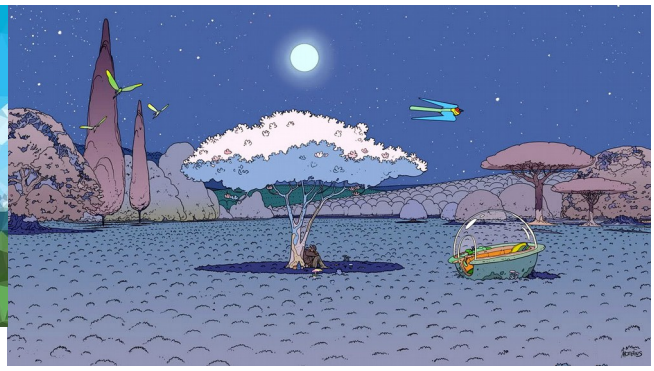
Mountain vector

create-flat-style-vector-map



Landscape-Flat-Design

Mountain Landscape Low poly



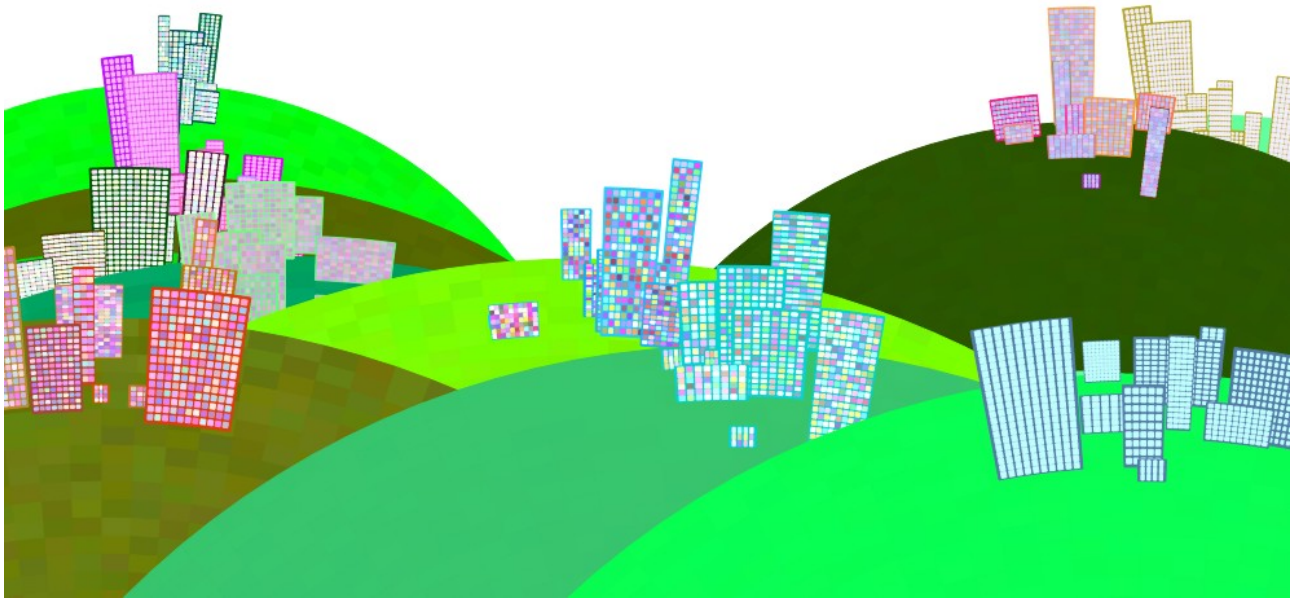
Moebius Landscape

Allez explorer d'autres univers, sous-marins, cosmiques, abstraits pourquoi pas... Pages suivantes j'utilise un code perso de génération de scènes pour illustrer concrètement les indications : ***le but du projet n'est pas d'imiter ce résultat particulier mais d'en extraire les principes***.

Exemple

Le générateur suivant a été fait pour tester la viabilité du projet. Il propose 3 plans ou calques, un arrière plan «ciel/nuages», un plan médian «urbain/collines», un avant plan «plaine/végétation» :

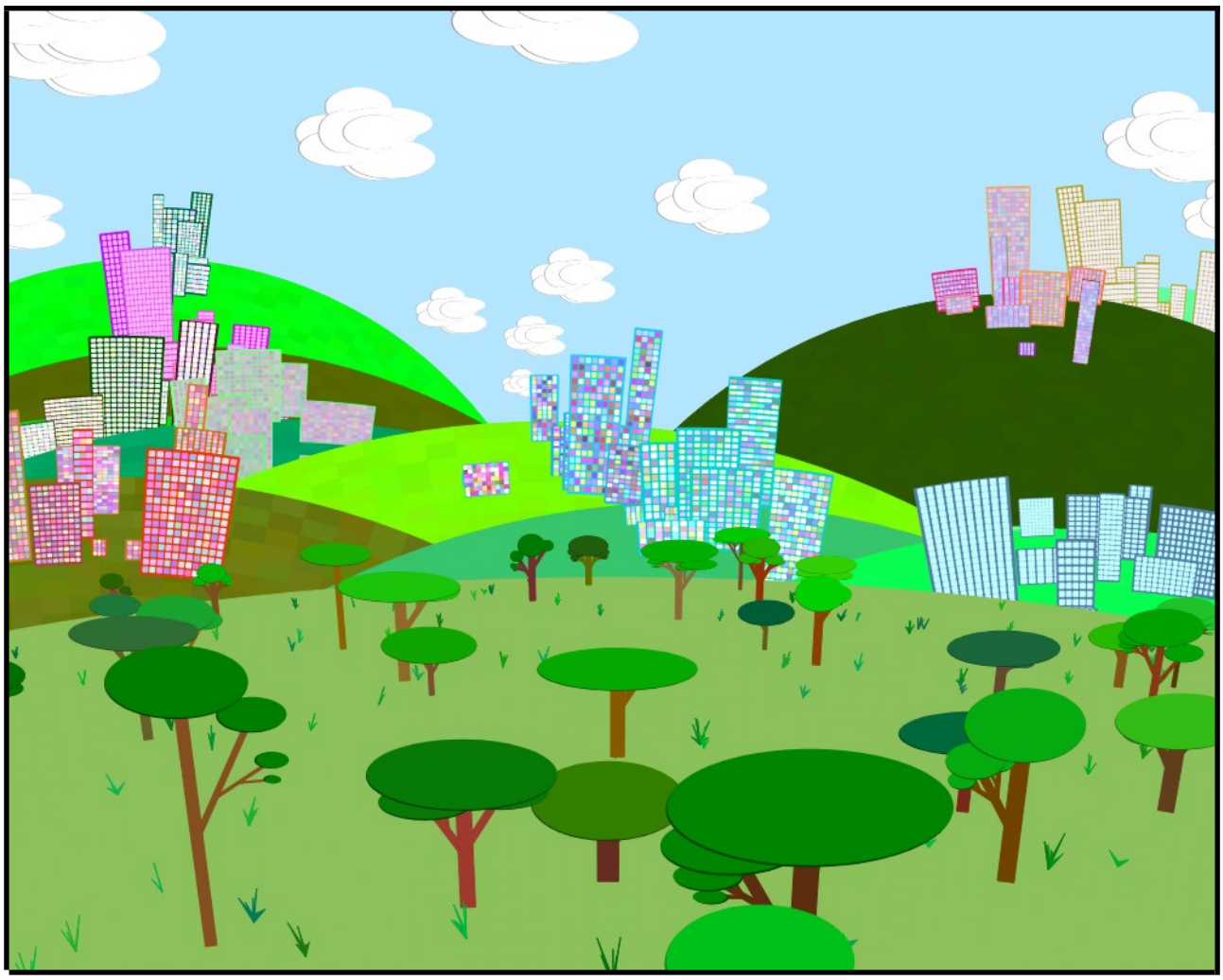
Arrière plan



Plan médian

Avant plan

Les 3 plans mis ensemble composent un paysage complet :



Sans quitter l'application, en modifiant des **paramètres** du générateur à la console, densité des herbes, couleurs dominantes pour les 3 plans, angles moyen des branches des arbres, courbure moyenne des collines, on obtient des paysages différents mais on reste sur le même schéma général.

Les ellipses n'étant pas fournies dans la bibliothèque Svgfile il a fallut ajouter cette primitive graphique grâce la documentation du format SVG. Mais ce n'est pas l'essentiel : le gros du travail est dans la modélisation de la hiérarchie plans/objets/primitives.

Générateurs pseudo-aléatoires et « graines » (seeds)

Pour ce projet il est indispensable d'apprendre à utiliser les générateurs pseudo-aléatoires de façon plus fine que d'habitude. La bibliothèque standard du C++ propose de nombreux moteurs de génération pseudo-aléatoire, dont l'usage est complexe et la diversité ne présente pas d'intérêt à notre niveau. En revanche ce qui est à retenir c'est que contrairement au C où nous avons un seul générateur global (rand) et une seule graine globale (srand) nous pouvons maintenant utiliser **plusieurs générateurs initialisés par plusieurs graines en parallèle**.

Un **objet générateur** est comme une **suite numérique définie par récurrence $u_{n+1}=f(u_n)$**, et la valeur donnée à la graine correspond à la valeur initiale **u_0**. Bien que les valeurs successives fournies par un générateur **ressemblent à du hasard** la séquence est **parfaitement déterministe** : **on obtiendra toujours la même séquence de valeurs quand on part de la même valeur initiale.**

Ceci est plutôt gênant quand on voudrait du « vrai hasard » et c'est la raison pour laquelle on initialise la graine avec une valeur tout le temps différente et « arbitraire » : le fameux time(NULL). Mais il y a 2 situations dans lesquelles on est content d'avoir une séquence déterministe qui dépend de la graine : quand on fait du débogage, pour trouver les conditions d'un plantage par exemple, et quand on fait du génératif et qu'on veut pouvoir régénérer les mêmes objets. C'est pourquoi je vous suggère de compléter la « bibliothèque » util avec ces 2 versions surchargées de alea :

```
/// Cette fonction retourne un entier aléatoire dans [min...max]
int alea(int min, int max, std::mt19937& randGen)
{
    return std::uniform_int_distribution<>(min, max)(randGen);
}

/// Cette fonction retourne un réel aléatoire dans [min...max[
double alea(double min, double max,  std::mt19937& randGen)
{
    return std::uniform_real_distribution<>(min, max)(randGen);
}
```

L'exemple d'utilisation suivant montre 2 objets générateurs (rg1 et rg2) initialisés avec des valeurs distinctes (0 et 1) et qui donnent 2 séquences « aléatoires » qui sont toujours identiques d'une exécution à l'autre. Pour changer de séquence aléatoire il suffit de changer ces valeurs initiales (soit en dur dans le code, soit par saisie utilisateur, soit par lecture d'un fichier) et de relancer la séquence d'utilisation (soit en relançant l'exécution, soit en ayant la séquence de génération dans une boucle et le générateur est re-déclaré à chaque fois). Testez pour bien comprendre...

```
void testAlea()
{
    int graine1=0;
    int graine2=1;

    std::mt19937 rg1{graine1};
    std::mt19937 rg2{graine2};

    for (int i=0; i<20; ++i)
        std::cout << util::alea(1, 6, rg1) << " ";
    std::cout << std::endl;

    for (int i=0; i<20; ++i)
        std::cout << util::alea(1, 6, rg2) << " ";
    std::cout << std::endl;
}
```

En mémorisant et en utilisant plusieurs graines il est possible de retrouver précisément les mêmes séquences de génération. Par exemple avec 3 graines associées chacune à un plan du générateur de paysage on peut permettre à l'utilisateur de renouveler un plan en particulier sans modifier les 2 autres. Exemple ci-dessous on change uniquement la graine pour le plan médian : l'avant plan (forêt) et l'arrière plan (ciel) sont re-générés strictement à l'identique.

Graine plan médian = 5

Graine plan médian = 75

Travail à réaliser, base sur 15 points

Vous proposerez un générateur paramétrique de paysages écrit en C++, pouvant utiliser le langage jusqu'à la version C++ 14 et les bibliothèques standards associées, ainsi que les codes précédemment distribués en TP sans aucune restriction.

Sans quitter l'application il sera possible de générer successivement plusieurs scènes, différentes, en fonction des paramètres entrés par l'utilisateur à la console à l'aide d'un menu. Les scènes générées seront visibles dans un navigateur en ouvrant le fichier output.svg.

L'utilisateur dispose **au minimum** de 3 paramètres pour chaque plan : la valeur de la **graine** servant de départ au générateur aléatoire du plan (voir exemple ci-dessus), et 2 autres paramètres, par exemple un paramètre de densité et un paramètre de taille moyenne. Exemple ci-dessous on a modifié la densité et la taille des arbres. Autres paramètres envisageables : couleur, intervalles de variation (variabilité), rapport largeur/hauteur (facteur de forme élongation) … liste non exhaustive.

Les scènes comporteront au moins 3 plans (ou « calques »), chaque plan proposera des objets graphiques plus ou moins complexes (nuages, maisons, arbres, cailloux, montagnes...) qui seront répartis de manière pseudo-aléatoire et auront eux même une diversité. Sur l'exemple précédent, les nuages sont clonés, mais chaque arbre et chaque immeubles est unique.

Dans chaque plan, l'ordre d'affichage des objets sera déterminé de telle sorte que les objets qui apparaissent visuellement devant cachent bien les objets qui apparaissent derrière. L'un au moins des plans sera composé de 2 catégories d'objets distincts. Sur l'exemple précédent à l'avant plan on voit à la fois des objets arbre et des objets herbes, et les objets dont la base est la plus basse en ordonnées sont affichés en dernier. Il y a des alternatives plus ou moins « astucieuses » mais la façon propre de réaliser ces 2 contraintes en même temps est d'utiliser le **polymorphisme** avec une classe de base abstraite dont dérivent les classes des objets qui pourront être mis dans un même conteneur et triés avec la fonction générique std::sort.

Sur vos images de paysages, on identifiera clairement des effets de translation qui se composent. Sur l'exemple précédent, une ville est construite avec des immeubles qui sont à des positions différentes, et les villes sont elles même à des positions différentes. Vous pouvez construire les objets directement à leur position définitive dans le **contexte** (dans le repère) de l'objet composite de plus haut niveau, ou laisser les objets définis à l'origine et ajouter les translations de position au fur et à mesure que les composites appellent les méthodes dessiner des objets composants (approche préférable, en particulier pour les rotations et homothéties). Dans un cas comme dans l'autre il faudra appliquer successivement les transformations, de façon cohérente.

Pour rappel le TD/TP 7 propose un code (lien en bas de la 1ère page de l'énoncé) couvrant l'ensemble des similitudes du plan (translations, rotations, homothéties, symétries). Il n'est pas nécessaire de faire le TD/TP 7 pour attaquer le projet mais il est recommandé de télécharger ce package et d'étudier le code client donné en exemple (main.cpp).

Enfin on veut un projet clairement orienté objet : aux objets graphiques présents à l'écran doivent correspondre des classes clairement identifiables dans le code. Les objets doivent exister sous forme d'instances, avec des relations d'appartenance logiques : les éléments des plans doivent se trouver dans des attributs conteneurs. Au cours 9 (semaine du 05/11) on présentera une façon générale de structurer des hiérarchies d'objets contenants/contenus : le « pattern composite ». Cette architecture avancée ne sera pas obligatoire mais **l'aspect « objet » sera vérifié lors de la soutenance : à résultat équivalent un projet ne respectant pas les principes de la programmation orientée objet se verra pénalisé** plus ou moins durement, un projet sans aucune classe étant à la limite du hors sujet pour ce module (donc à la limite du 0).

Extensions, 5 points maximum

Les propositions d'extensions qui suivent n'ont de sens que si le CDC de base est très largement rempli. Le jury a le droit de ne pas compter de points supplémentaires sur des extensions si l'implémentation du CDC de base est trop faible : ceci traduirait une mauvaise gestion des priorités. Les points seront attribués selon la difficulté des extensions, la pertinence des techniques de programmation objet utilisées, et la qualité de leur réalisation. Il n'est évidemment pas nécessaire de faire **toutes** ces extensions pour avoir 20/20 ! Au delà de 5 points la note d'extension sature.

→ Sauvegarde des jeux de paramètres (incluant les graines) pour pouvoir retrouver des scènes
→ Accès aux objets générés en mode « édition », par exemple on peut désigner un des éléments du paysage généré et le déplacer, ou l'effacer, sans toucher au reste
→ Effets de perspective : utiliser les homothéties pour faire comme des points de fuite
→ Effets de déformations (pas juste les similitudes du plan) courbures, ondulations, zigzags...

→ Compositions de rotations/transformations complexes successives
→ Maillages évolués (quads, polygone quelconques...)
→ 3D réelle (x,y,z) ce n'est pas si compliqué que ça quand on a déjà un système de maillage. La projection 3D → 2D avec perspective est très simple : $x_{2D} = x/z$ $y_{2D} = y/z$
Les rotations dans l'espace sont plus complexes, adapter les classes Matrice et Transformation...
→ Compléter les primitives SVG (ellipses...)
→ Compléter les couleurs SVG : transparences, gradients...
→ Faire une/des catégories d'objets « arborescents » avec des branches clairement visibles et des divisions successives (branches, sous-branches...).
→ Plus de plans que le minimum demandé
→ Plus de catégories d'objets que le minimum demandé
→ Grande diversité dans les scènes (des fois montagne, des fois plaine, des fois maritime...)
→ Hiérarchie générative complexe : sur l'exemple les villes du plan médian sont parfois homogènes parfois diverses... L'idée est d'augmenter le **contraste aléatoire** à l'intérieur d'une scène et entre scènes en définissant (aléatoirement) des sous-groupes d'objets avec des moyennes et des variances spécifiques.

```
17 0 -> 17 17 17 17 17 17 17 17 17 17
15 1 -> 14 15 14 14 15 15 16 16 16 14
19 0 -> 19 19 19 19 19 19 19 19 19 19
17 5 -> 12 20 18 14 22 21 14 13 18 20
25 5 -> 21 24 25 24 28 23 27 22 25 21
```

```
void testAleaHierarchique()
{
    int graine=7; // Graine arbitraire
    std::mt19937 rg{graine};

    for (int i=0; i<5; ++i)
    {
        // 1 chance sur 2 d'avoir une variance nulle
        // 1 chance sur 2 d'avoir une variance entre 1 et 5
        int var = std::max( 0, util::alea(-4, 5, rg) );
        int val = util::alea(15, 25, rg);
        std::cout << val << " " << var << " -> ";
        for (int j=0; j<10; ++j)
            std::cout << util::alea(val-var, val+var, rg) << " ";
        std::cout << std::endl;
    }
}
```

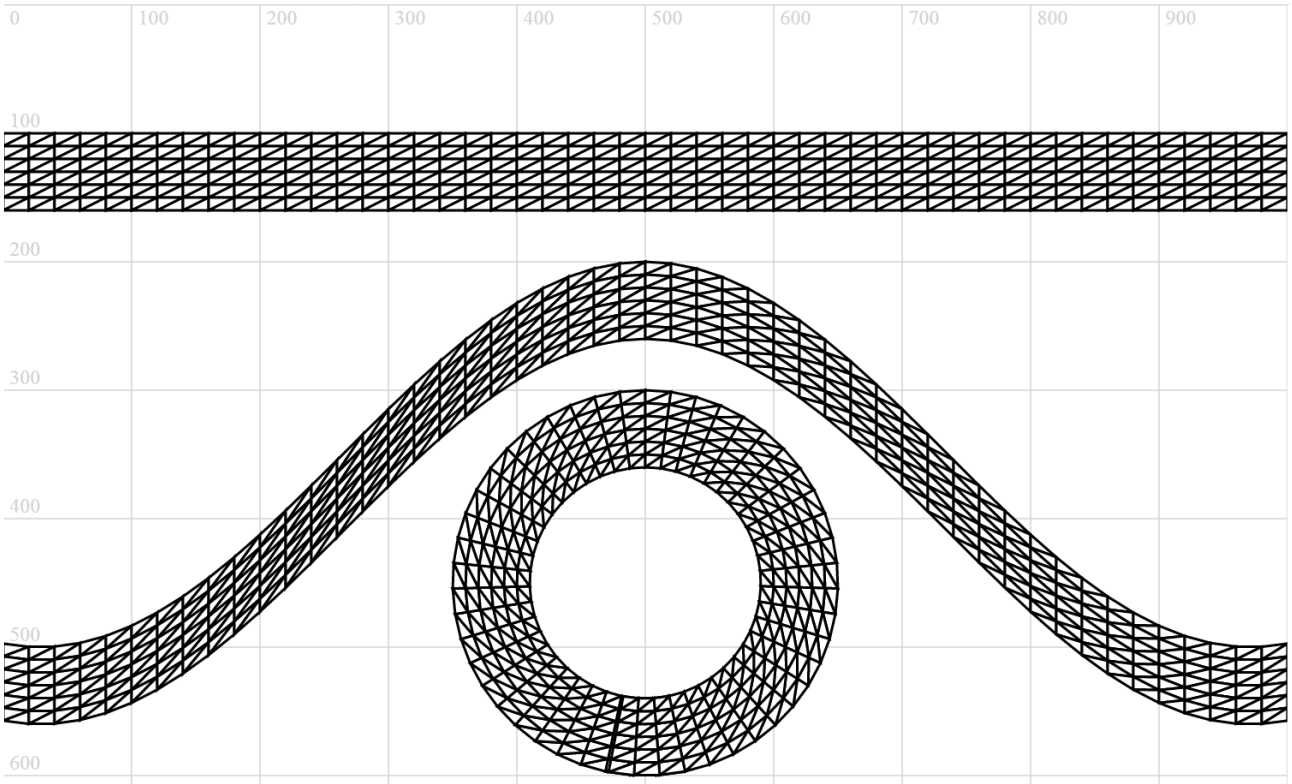
→ Toute autre idée qui vous semble intéressante, à discuter avec votre chargé de TP !

Conseils, qualité de l'approche objet

Quand vous développez le code de niveau application, demandez vous comment vous pouvez utiliser/compléter les classes de bases (les composants, les classes de bas niveau) pour qu'elles vous rendent service. Un objet Arbre a besoin d'être positionné en (x,y) ? Utilisez le type Coords. Vous avez souvent besoin de tirer aléatoirement dans le même intervalle ? Mettez en place une classe Intervalle avec une méthode alea (et un `std::mt19937& randGen` en paramètre). Besoin de souvent tirer aléatoirement un couple de valeurs ? Compléter la classe Cadre avec une méthode alea (et un `std::mt19937& randGen` en paramètre). Construire une application C++ c'est avant tout commencer pas construire le vocabulaire avec lequel on travaille, et compléter ce vocabulaire en fonction des besoins. Besoin fréquent de niveau application => compléter les classes de base.

A titre indicatif, le projet donné en exemple fait à peu près 2200 lignes de code (*code only*) dont 700 ont été reprises de la base de code du TD/TP 7. Il est donc attendu que vous écriviez entre 1500 à 2000 lignes de code pour viser un 20/20. Ce n'est qu'une indication bien sûr car tout dépend de la qualité du code et du résultat...

Et enfin soyez réalistes et rationnels dans votre processus de développement. Les collines comme sur les exemples montrés, c'est sympa mais ça implique de maîtriser des déformations avec des courbures. Avant de commencer à faire à la fois des immeubles (ou des maisons, ou des arbres) et des courbures de terrain, commencez déjà par mettre en place et valider les déformations sur des grilles simples. Et donc commencez d'abord par avoir un générateur de grilles !



Exemple de test préliminaire : validation de déformations sur des grilles simples...

Consignes générales

En dehors des exemples du cours et des codes officiellement fournis, toute détection de copie massive de code tiers sera considéré comme plagiat et très sévèrement sanctionné, avec circonstances aggravantes si
- l'emprunt n'est pas cité
- l'emprunt vient du travail d'une autre équipe d'étudiants de l'ECE pour ce même projet
- il y a tentative de dissimulation (maquillage de code...)

Le code source doit être présentable : bien indenté, raisonnablement aéré et commenté (chapitré). Les identifiants doivent être explicites, on doit comprendre de quoi on parle en lisant le nom des classes, des attributs, des méthodes. Le projet est structuré en .cpp et .h par classe et si possible hiérarchisé en arborescence de répertoire avec des thématiques (voir exemple code TD/TP 7).

Le projet est à faire en trinôme ou binôme, à l'intérieur d'un même groupe de TP. Les consignes de constitution des équipes de projet seront communiquées très rapidement de même que la date de soutenance et les modalités de rendu.

Robin Fercoq et toute l'équipe encadrante,
29/10/2018
Version 1, sujette à ajustements mineurs en cas de besoin.