Projet Prep'ISIMA :

Terrain 3D en Réalité Virtuelle

Enseignant responsable : Emmanuel MESNARD

PITO Thomas

CONNE Vincent

2020 - 2021

Remerciements

Avant tout, nous souhaitons remercier l'ensemble des personnes qui ont permis la réalisation de ce projet. Nos remerciements se tournent donc vers l'Institut Informatique d'Auvergne (anciennement nommée Institut Supérieur d'Informatique, de Modélisation et de leurs Applications - ISIMA) qui nous permet lors de notre cursus la réalisation de ce projet. Nous remercions également les enseignants qui nous ont entourés pour ce projet et en particulier E. MESNARD qui nous a suivis tout le long de celui-ci malgré les difficultés face aux contraintes sanitaires.

Nous remercions enfin, nos proches qui nous ont apporté leur soutien et leur aide tout le long de ce projet et notamment lors de la phase de test.

Résumé

Le cursus de classe préparatoire à l'école d'ingénieur en Informatique ISIMA, impose lors de sa deuxième année la réalisation d'un projet en binôme. Après proposition des différents sujets par Mme. LAGOUTTE, nous avons obtenu le sujet proposé par M. MESNARD portant sur la Réalité Virtuelle. L'objectif du projet était alors de générer un terrain 3D et de le visualiser dans un casque immersif, le tout par le biais du logiciel Processing. Le sujet laissant place à l'imagination, nous avons fait le choix de créer un monde 3D dynamique avec des textures issues du jeu vidéo "Minecraft". Pour rendre l'immersion plus intéressante, nous avons offert la possibilité à l'utilisateur de se déplacer sur le terrain mais également de changer le terrain 3D grâce à son regard. En effet, l'utilisateur peut générer 4 types de mondes différents : le monde lave, le monde neige, le monde nature, et le monde candy.

Enfin, la conception et la réalisation de ce projet furent entièrement réalisées en binôme, permettant à chacun de renforcer ses capacités de travail en équipe mais également ses aptitudes organisationnelles.

Dans le but d'offrir à chacun une meilleure compréhension de notre projet, vous trouverez ci-dessous une vidéo de présentation de celui-ci :

https://www.youtube.com/watch?v=3O0yUAswJL4&ab_channel=VincentConne

Sommaire :

Introduction

Dans le cadre de notre cursus en Prep'ISIMA, nous avons dû réaliser le projet suivant sur le domaine de la Réalité Virtuelle :

Titre du Sujet : *Terrain 3D en Réalité Virtuelle*

Environnement de développement :
Matériel : Ordinateur (PC ou Mac), et éventuellement, Smartphone Android + Casque immersif

Logiciel :

Processing 3.5.4 (https://processing.org/download/ , https://android.processing.org/).
(Processing est l'outil de développement employé durant le cours de « Réalité Virtuelle »)

Description

L'objectif de ce projet est de concevoir une application de Réalité Virtuelle sur un ordinateur, et qui pourra être ensuite portée sur un smartphone placé dans un casque (type cardboard) pour un rendu immersif stéréoscopique.
Un casque immersif « VR Elegant » sera mis à disposition pour cette réalisation.

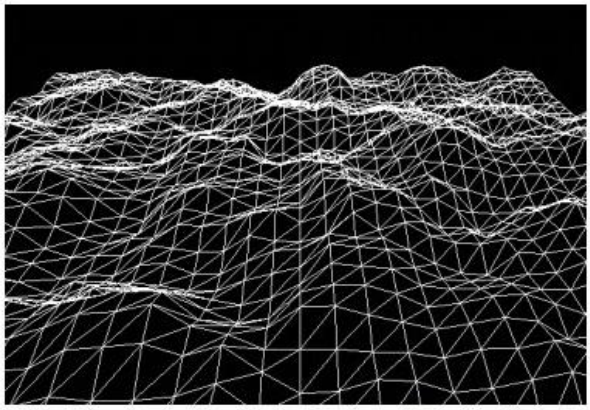
Casque immersif VR Elegant

A noter que cette application servira (très certainement !) aux diverses démonstrations et journées portes ouvertes de l'Isima...

L'idée générale de l'application consiste à **visualiser un terrain 3D**, généré de manière dynamique. Pour cela, il est possible (et conseillé !) de s'inspirer des générateurs en bruit de Perlin (« Perlin Noise ») déjà développés en Processing :

https://www.openprocessing.org/sketch/619145
https://www.openprocessing.org/sketch/584651
https://www.openprocessing.org/sketch/541783

Au minimum, le rendu sera de type « fil de fer », en **stéréoscopie** pour la version smartphone.



Capture du rendu d'un générateur de terrain en « Perlin Noise » et « fil de fer »

Bien sûr, tout rendu plus réaliste (couleur, texture, objets 3D, etc...) dans la génération du terrain sera particulièrement apprécié...

Capture du rendu « Perlin Noise » du générateur de terrain « MineCraft »

Ce document constitue ainsi un rapport complet sur ce projet en abordant ses différents aspects : conception, fonctionnement, et utilisation.

I - Qu'est-ce que la Réalité Virtuelle ?

La Réalité Virtuelle (Virtual Reality en anglais) désigne un ensemble de dispositifs permettant de simuler numériquement un environnement par le biais d'une machine telle qu'un ordinateur. La Réalité Virtuelle permet alors à l'utilisateur de se retrouver dans un monde généré numériquement. Les sensations ressenties par l'utilisateur peuvent prendre différentes formes : la vue est le sens le plus souvent utilisé, mais il est aussi possible de donner une impression d'immersion bien plus réaliste en ajoutant des effets sur d'autres sens tel que le toucher par exemple, où l'on peut très bien imaginer projeter de l'air sur l'utilisateur si celui-ci est dans un monde simulant sa présence au sommet de l'Everest. Ainsi, la Réalité Virtuelle permet à l'utilisateur de vivre une expérience d'immersion avec une activité senso motrice dans un monde artificiel. Pour favoriser l'immersion, l'utilisateur se sert d'un casque de Réalité Virtuelle qui utilise le principe d'affichage en 3D stéréoscopique pour permettre à son porteur de voir le monde virtuel comme s'il y était.

Figure 1 - Casque de Réalité Virtuelle Playstation

La Réalité Virtuelle représente ainsi un potentiel énorme pour les années futures car celle-ci peut être utilisée dans de nombreux domaines que ce soit pour la formation de personnels (moins coûteux et moins risqué), pour la maintenance et l'assistance, pour la supervision, pour le médical, pour le marketing, pour la simulation ou encore pour le domaine culturel.

II - Présentation de Processing et des bibliothèques utilisées

A. Présentation de Processing

Figure 2 - Logo Processing

Processing est une bibliothèque graphique gratuite et un environnement de développement intégré (IDE) conçu pour les communautés d'arts électroniques, d'art des nouveaux médias et de conception visuelle dans le but d'enseigner aux non-programmeurs les principes de base de la programmation informatique dans un contexte visuel. Le traitement utilise le langage Java, avec des simplifications supplémentaires telles que des classes supplémentaires, des fonctions et opérations mathématiques. Il fournit également une interface utilisateur graphique pour simplifier la phase de compilation et d'exécution.

B. Bibliothèques utilisées

Afin de mener à bien notre projet, nous avons utilisé 2 bibliothèques de Processing, respectivement la bibliothèque **video** et la bibliothèque **vr**.

La bibliothèque **video** intégrée dans Processing lit les fichiers vidéo et capture les données vidéo d'une caméra. La vidéo peut être capturée à partir de caméras USB, de caméras IEEE 1394 (Firewire) ou de cartes vidéo avec des périphériques d'entrée composite ou S-vidéo connectés à l'ordinateur. Les films peuvent être chargés à partir de fichiers situés sur l'ordinateur ou n'importe où sur Internet. Cela est basé sur le framework multimédia GStreamer et utilise les liaisons gstreamer-java pour interfacer GStreamer à partir de Java afin de prendre en charge un large éventail de formats multimédias.

La bibliothèque **vr** intégrée dans Processing quant à elle, comprend deux moteurs de rendu : VR (ou STEREO) et MONO. Le rendu VR / STEREO génère une vue stéréo de l'esquisse, rendant chaque image deux fois (une fois pour chaque œil) et appliquant les transformations oculaires appropriées.

Le moteur de rendu VR / STEREO que nous utilisons fonctionne en dessinant chaque image deux fois, une fois pour chaque œil.

III - Présentation du projet

A. Présentation du projet attendu

Pour commencer, l'objectif du projet est de réaliser un terrain 3D dynamique en réalité virtuelle sur lequel on peut se déplacer. Par ailleurs, ce dernier doit être portable sur smartphone afin d'obtenir un rendu immersif du terrain sur lequel nous nous déplaçons par le biais d'un casque de Réalité Virtuelle.

B. Présentation du rendu final

Le rendu final est concluant et répond à la problématique énoncée. En effet, nous avons généré un terrain dynamique 3D sur lequel l'utilisateur peut se déplacer. Notre terrain est composé de plusieurs modes correspondant à des mondes différents. Ces derniers sont au nombre de 4 et constituent respectivement les modes nature, lave, neige, et candy. Ils sont alors différenciés par leurs textures à chaque génération. Cette génération s'effectue grâce à la mise en place de 4 sphères positionnées autour du terrain. A chaque fois que le regard se pose sur une de ces 4 sphères externes au terrain, durant un certain laps de temps correspondant à quelques secondes, le terrain se régénère par rapport au monde qu'entraine la sphère en question. Le joueur voit alors le monde changer autour de lui, lui permettant ainsi de se déplacer dans le nouveau monde généré.

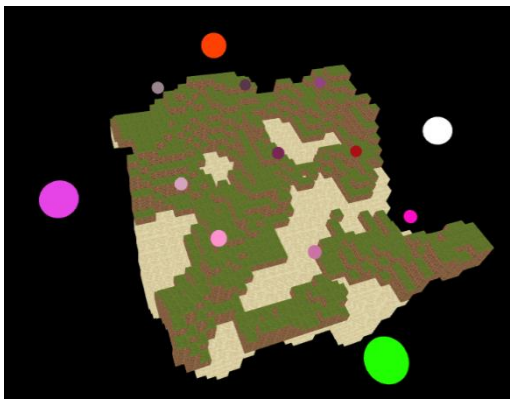


Figure 3 - Rendu final vu de haut

Pour ce qui est des déplacements, nous sommes parvenus à utiliser des sphères de la même manière que pour les sphères extérieures. En effet, 9 sphères internes sont placées dans l'espace et à chaque fois que le regard de l'utilisateur se pose sur une de ces 9 sphères internes au terrain, durant un certain laps, le terrain se déplace alors, laissant croire par illusion à l'utilisateur qu'il se déplace. Ainsi, le joueur est « téléporté » à l'emplacement de la sphère qu'il regardait.

IV - Implémentation du projet

Au sein de cette partie, nous allons vous présenter l'implémentation concrète du projet. En effet, nous verrons les différentes structures utilisées et les choix architecturaux correspondants.

A. Le terrain 3D

Tout d'abord, pour mener à bien ce projet, il fut essentiel d'avoir la capacité de générer dynamiquement un terrain 3D et de l'afficher à l'utilisateur. Ci-dessous, nous allons vous expliquer les différentes structures utilisées pour construire le terrain mais également les différentes fonctions afférentes ainsi que les textures appliquées sur celui-ci.

- La structure du terrain :

Pour réaliser notre projet, il est évident que nous avions besoin de générer un terrain 3D dynamique. La première question qui s'est alors posée était de savoir comment stocker ce terrain et qu'elle serait la forme de celui-ci. Ainsi, nous avons fait le choix de créer un terrain carré avec une dimension de 30 cubes de largeur et de 30 cubes de longueur (on a également choisi une taille de 50 pour les cubes). On notera que nous avons également fait le choix de générer notre terrain sous forme de cubes pour obtenir un visuel proche du jeu vidéo "Minecraft"[1] qui nous avait été abordé comme exemple par M. MESNARD. De plus, nous avons choisi d'imposer une hauteur maximale à notre terrain 3D pour éviter de générer un nombre trop important de cubes ce qui aurait diminué les performances de notre programme. Pour ce qui est du stockage de notre terrain, nous avons fait le choix de le stocker sous la forme d'un tableau à deux dimensions et d'un tableau à trois dimensions. En effet, nous verrons dans la partie suivante qu'un générateur en "Perlin Noise" nous a permis d'obtenir un terrain 3D que nous avons stocké dans un tableau à deux dimensions. Puis, cette structure de tableau nous a permis

Figure 5 - Monde Minecraft

[1] Minecraft est un jeu vidéo de type « bac à sable » développé par Markus Persson, puis par la société Mojang Studios.

facilement de convertir le tableau à deux dimensions en un tableau à trois dimensions pour ainsi obtenir notre structure finale représentant le terrain dynamique 3D. Les différentes dimensions du terrain ont été choisies après différents tests afin de garantir un programme le plus fluide possible lors de son utilisation.

- Le noise, créateur du terrain :

Voici une des parties les plus intéressantes de notre projet. En effet, un des objectifs de celui-ci était de générer un terrain 3D dynamique et donc d'avoir une génération dite **aléatoire**. Pour ce faire, nous avons utilisé un générateur de terrain aléatoire en "Perlin Noise". En effet, le "Perlin Noise" est à l'origine une texture procédurale primitive. Il s'agit d'un bruit de gradient utilisé pour améliorer le réalisme des infographies. La fonction a un aspect pseudo-aléatoire, cependant il est possible, par le biais de propriétés, de plus ou moins contrôler son aspect aléatoire. Le bruit de Perlin est alors souvent utilisé dans les images de synthèse pour la création d'éléments tels que le feu, la fumée ou encore les nuages. En effet, le bruit de Perlin permet de générer aléatoirement des éléments en conservant une certaine **uniformité**. C'est pourquoi, l'utilisation du bruit de Perlin dans notre projet est très utile car c'est ce bruit de Perlin qui permet de générer **aléatoirement** un terrain 3D de façon uniforme. Ainsi, avec une génération en bruit de Perlin, cela évite d'obtenir un terrain irréaliste avec des incohérences de génération.

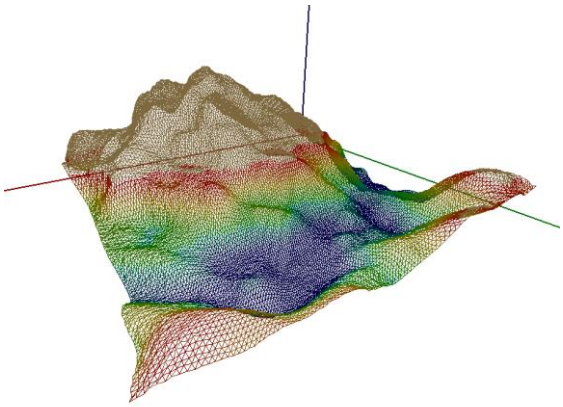


Figure 6 - Perlin Noise

Pour générer notre terrain, nous avons donc utilisé un algorithme de bruit de Perlin que nous avons nommé dans une fonction "noiseSpace" qui nous permet en prenant comme paramètre un flottant "flying" de remplir un tableau à deux dimensions représentant notre terrain 3D. Le tableau est alors rempli de la manière suivante : à chaque case de celui-ci, on insère une valeur comprise entre 0 et 1 qui représentera par la suite la hauteur du terrain à ces coordonnées. Cette valeur est alors déterminée par la fonction **noise()** qui correspond à la génération en bruit de Perlin. Cette génération avec la fonction **noise()** prend alors parmi ses paramètres le "flying" qui correspond à la position de déplacement du terrain (flying = 0 représente le "départ" du terrain, et plus le flying augmente, plus on avance sur le terrain). Ainsi, après que toute la structure de tableau à deux dimensions ait été complétée

par le générateur en bruit de Perlin, nous disposons d'un tableau à deux dimensions représentant le terrain. Enfin, le choix d'utiliser la variable "flying" s'explique par notre volonté de pouvoir changer le terrain par la suite. En effet, lorsque l'on souhaitera regénérer le terrain dans un nouveau monde, il suffira simplement de changer la valeur du flying, ce qui nous permettra ainsi d'obtenir une nouvelle génération de terrain. De plus, nous avons fait le choix d'utiliser un tableau à deux dimensions après génération par le bruit de Perlin, car cela facilite notre compréhension du fonctionnement du bruit de Perlin, et nous permet de différencier le tableau obtenu par ce bruit, et celui qui représente réellement notre terrain 3D.

- La conversion 2D vers 3D :

A présent, nous disposons d'un tableau à deux dimensions représentant notre terrain, mais cela signifie que pour chaque case représentée par un couple (x,y), il y est stocké une valeur entre 0 et 1. Cependant, cette représentation structurelle ne nous semblait pas assez explicite pour notre terrain dynamique 3D. C'est pourquoi nous avons fait le choix d'effectuer une conversion de ce tableau à deux dimensions en un tableau à trois dimensions. En effet, une case serait à présent représentée par un trio (x,y,z) où z représente la hauteur, hauteur déterminée par ce qui était stocké dans chaque case du tableau à deux dimensions. Mais alors qu'est ce qui est stocké dans chaque case de ce nouveau tableau ? Le principe est simple : si à cet emplacement il doit y avoir un cube du terrain 3D alors la case contient un 1, sinon la case contient un 0. Ainsi, on se retrouve avec une structure de tableau à trois dimensions représentant un terrain 3D, qui nous indique pour chaque emplacement du terrain s'il doit y avoir un cube ou non.

Remarque : vous noterez dans le code, que nous avons fait le choix d'effectuer un ajout automatique et un ajout manuel de cubes. En effet, le problème d'une génération en Perlin Noise où l'on ne prend en charge que la surface du terrain est qu'il se peut qu'il y ait des falaises pour lesquelles les cubes ne sont pas tous générés. Voulant obtenir un rendu le plus fluide possible pour l'utilisateur, il n'a pas été possible de générer tous les cubes sous la surface du terrain car cela aurait causé trop de "lag". C'est pourquoi, un ajout automatique[2] et un ajout manuel[3] d'un certain nombre de cubes sous la surface sont effectués pour pallier à ce problème, et ainsi garantir la fluidité.

[2] L'ajout automatique est calculé par une opération dépendante des dimensions initiales.
[3] L'ajout manuel a été déterminé après différents tests pour garantir une fluidité du programme.

- Le rendu du terrain :

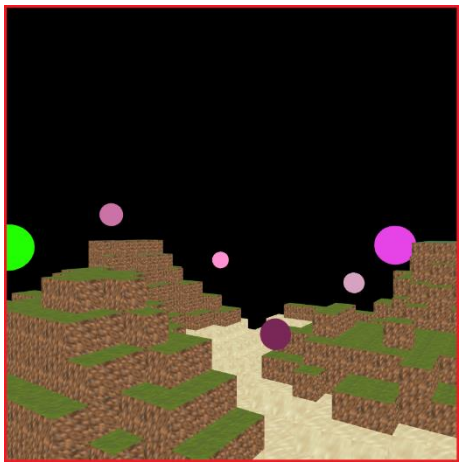


Figure 7 - Rendu du terrain (Nature)

Maintenant que nous disposons de notre structure représentant notre terrain dynamique 3D, intéressons-nous à son dessin et son rendu. Avant toute chose, il faut comprendre que le programme a pour but de permettre à l'utilisateur de se déplacer sur le terrain. Mais attention : l'utilisateur ne se déplacera pas réellement, au contraire, c'est le terrain qui va être déplacé donnant ainsi l'illusion de déplacement à l'utilisateur. Ce déplacement du terrain va alors être assuré par la commande **translate()** qui va permettre de changer l'emplacement de dessin du terrain. On notera que les coordonnées de ce translate sont représentées par les variables "posTranslateX" et "posTranslateY" qui sont initialisées à -4 et -8 pour centrer le terrain autour du joueur au début du programme. A présent, pour dessiner le terrain nous avons fait le choix de créer une fonction nommée "DessineTerrainMode" qui effectue le dessin du terrain selon le mode choisi. On rappelle qu'il existe 4 mondes et donc 4 modes de terrain (0=Nature, 1=Lave, 2=Neige, 3=Candy). Ainsi, la fonction parcourt l'intégralité du tableau à trois dimensions et si à ces coordonnées est stocké un 1, alors on se déplace aux coordonnées correspondantes par le biais d'un translate pour dessiner le cube. On remarquera que le cube dessiné et plus particulièrement sa texture dépendent d'une part de la hauteur à laquelle celui-ci doit être dessiné et d'autre part du mode du terrain. Ainsi, après exécution de cette fonction, nous obtenons un rendu complet du terrain dynamique 3D.

- Les textures des cubes :

Pour terminer sur la génération du terrain, nous avons fait le choix d'appliquer des textures au terrain et plus particulièrement aux cubes de celui-ci pour gagner en réalisme et en immersion. Pour ce faire, nous avons utilisé deux textures pour chaque type de terrain. La première est appliquée si le cube est en hauteur alors que la seconde est appliquée si le cube a une faible altitude sur le terrain. Le chargement et l'application de ces textures sont réalisés par la fonction "setupCreationCubesTextures" qui est appelée dans la partie **setup** du programme.

Enfin, il est important de préciser que ces textures ne sont pas notre propriété[4]. Celles-ci ont été extraites du célèbre jeu vidéo cubique "Minecraft" dont les textures sont en accès libre. Leur utilisation nous

[4] Les textures sont la propriété de la société Mojang Studios : https://www.minecraft.net/fr-fr/

a ainsi semblé intéressante pour mettre en lumière un monde virtuel 3D cubique tel que le fait l'entreprise Microsoft depuis 2014.

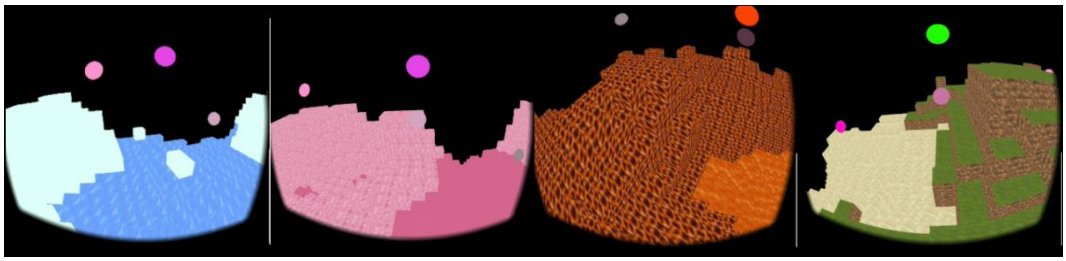


Figure 8 - Les différentes textures des mondes

B. Les sphères internes

Par la suite, il fallait trouver un moyen de nous déplacer au sein même du terrain 3D. Nous allons donc vous expliquer les différentes structures utilisées pour effectuer le déplacement du joueur mais également les différentes fonctions associées.

- Les différentes structures :

Pour commencer, le but est de savoir comment se déplacer avec une certaine facilité. Pour ce faire, nous avons utilisé des sphères, placées harmonieusement au sein du terrain. En effet, l'utilisation de ces dernières permet de déplacer le terrain sous le joueur et ainsi de créer l'illusion que le joueur se déplace dans l'espace à l'emplacement de la sphère. Nous avons alors utilisé deux tableaux d'entiers qui contiennent la position prédéfinie des sphères internes sur les "coordonnées Translate" X et Y. Ces deux tableaux permettent alors de connaître les coordonnées du translate à effectuer pour se déplacer sur une sphère en effectuant un déplacement du dessin du terrain. Nous utilisons également trois tableaux permettant de générer les sphères internes et de connaître pour chaque sphère interne sa couleur en RGB (ces couleurs sont prédéfinies). La connaissance de la couleur de chaque sphère interne sera par la suite utilisée pour savoir quelle sphère l'utilisateur regarde et ainsi pouvoir effectuer le déplacement sur cette sphère interne.

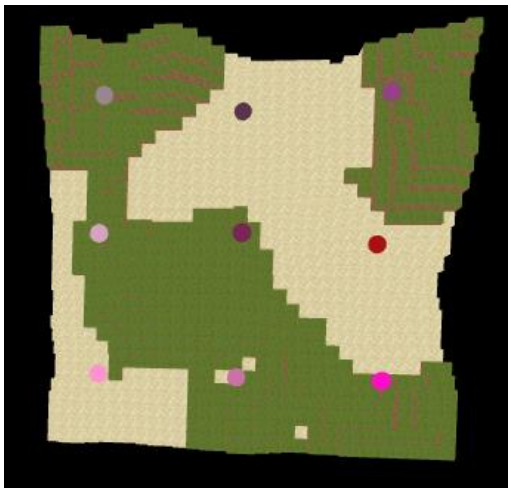


Figure 9 - Vue aérienne des 9 sphères internes

De plus, nous avons créé un tableau d'entiers qui contient pour chaque sphère interne, un compteur permettant de savoir si celle-ci est regardée depuis plus ou moins longtemps par le joueur.

Enfin, pour gagner en praticité de code, on connaît également, grâce à une variable, la position actuelle de l'utilisateur sur le terrain grâce au numéro de sphère interne sur laquelle il est positionné.

- Les fonctions des sphères internes :

Nous avons alors utilisé différentes fonctions afin de mener à bien l'utilisation de ces sphères internes. Pour commencer, il nous faut une fonction permettant de dessiner une sphère. Dans celle-ci, on se déplace aux coordonnées prises en paramètre que l'on multiplie par la taille des cubes et on ajoute 200 à notre hauteur afin de positionner les sphères au-dessus de la surface du terrain et non pas dans les cubes du terrain. Cela permet alors un affichage des sphères correcte qui n'entrent pas en conflit avec les cubes déjà dessinés à cette position. Pour effectuer cela avec l'ensemble des sphères, nous utilisons une tierce fonction. A chaque parcours des tableaux de position, on switch sur la position actuelle de la sphère, donc du joueur, pour pouvoir uniquement dessiner les sphères internes qui l'entourent. Pour finir, durant le même parcours, on remplit les tableaux qui contiennent les positions des sphères internes sur les différents axes X, Y, Z (coordonnées du tableau à trois dimensions).

- Les déplacements sur le terrain 3D :

Il nous a fallu du temps pour comprendre la gestion des mouvements via l'utilisation des sphères. Pour que l'utilisateur se déplace sur le terrain, nous avons fait le choix que les déplacements soient effectués par regard de l'utilisateur. En effet, si l'utilisateur regarde fixement durant quelques secondes une sphère interne alors il sera déplacé sur celle-ci. Nous avons alors utilisé deux fonctions permettant d'effectuer le déplacement de l'utilisateur sur une sphère interne. La première fonction a pour but de détecter si l'utilisateur regard une sphère interne ou non. Son fonctionnement est alors le suivant : on parcourt l'intégralité des pixels centraux de l'image, et si un pixel porte la couleur d'une des sphères internes, alors on incrémente le compteur de regard de cette sphère dans le tableau correspondant. La seconde fonction permet quant à elle d'effectuer le déplacement sur la sphère. En effet, lorsque le compteur de regard d'une sphère interne atteint la valeur 20 (prédéfinis), alors cela signifie que l'utilisateur a bien regardé fixement cette sphère et qu'il souhaite s'y déplacer. On change alors la sphère actuelle sur laquelle le joueur se situe, puis on réinitialise les compteurs de regard des sphères internes (et externes) grâce à une fonction auxiliaire afin de réinitialiser complètement le regard du joueur. Pour finir, on effectue des changements sur les

positions de translate : on change donc la position actuelle de translate (pour le dessin du terrain) sur les différents axes par celle correspondant à la sphère interne. Ces changements permettent alors de dessiner le terrain de façon que le joueur soit bien positionné sur celui-ci après son pseudo déplacement.

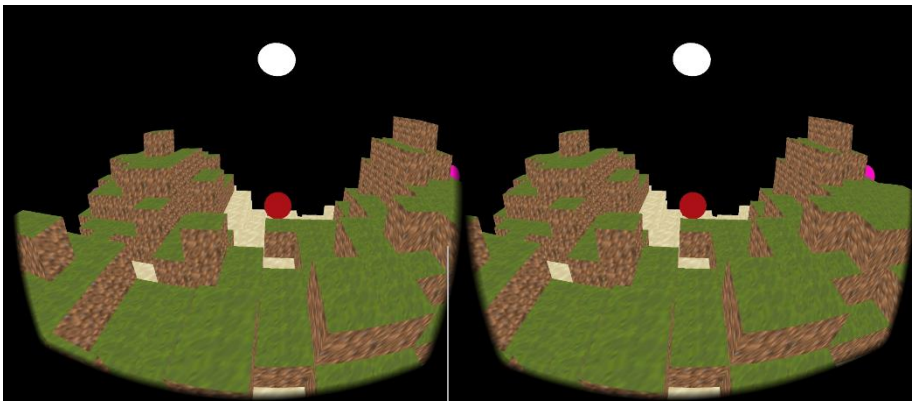


Figure 10 - Utilisateur observant une sphère interne au centre de sa vision (couleur rouge)

C. Les sphères externes

- Structures des sphères externes :

A l'instar des sphères internes, les sphères externes possèdent différentes structures qui nous permettent de les utiliser pour régénérer le terrain selon le mode choisi par l'utilisateur (nature, neige, lave, candy).

Ainsi, les sphères externes possèdent 3 structures principales sous forme de tableaux d'entiers à une dimension. La première structure représente la couleur des sphères externes en RGB. Elle est constituée de trois tableaux nommés respectivement "CouleurSpheresRE", "CouleurSpheresGE" et "CouleurSpheresBE". Les couleurs en RGB sont définies au préalable pour ainsi offrir une connaissance parfaite et permanente des couleurs de ces sphères externes. Cette information de couleur de ces sphères externes permettra par la suite de pouvoir détecter si l'utilisateur regarde une de ces sphères externes et souhaite donc regénérer le terrain selon le mode. Ces 4 sphères ont donc des couleurs liées au monde qu'elles engendrent : vert pour nature, rouge orangé pour lave, blanc pour neige, et rose pour candy. La seconde structure de ces sphères externes est un tableau d'entiers contenant leur position prédéfinie en coordonnées "sketch" classique. On retrouve ainsi 3 tableaux d'entiers à une dimension nommés respectivement "PosSpheresEXTX", "PosSpheresEXTY" et "PosSpheresEXTZ" qui stocke la position de ces sphères sur les différents axes X, Y et Z. Enfin, la troisième structure est un tableau d'entiers à une dimension qui

permet de stocker pour chacune des 4 sphères externes un compteur permettant de savoir pour chacune d'elles si elle est regardée depuis plus ou moins longtemps par l'utilisateur.

Remarque : ces 4 sphères sont positionnées sur les 4 côtés du terrain et ont une taille plus importante que celle des sphères internes pour faciliter leur différenciation.

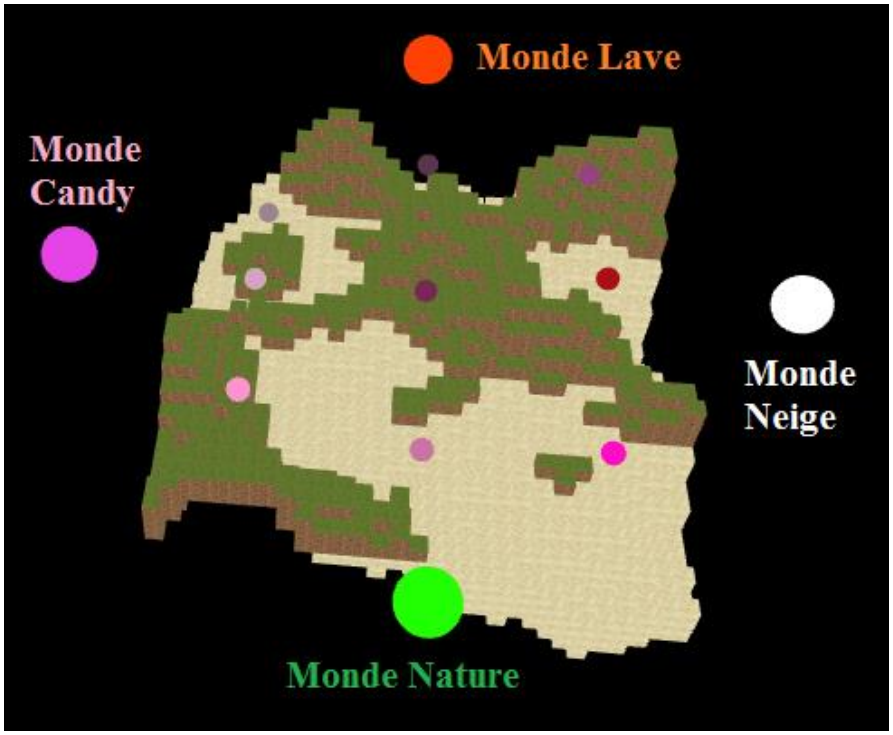


Figure 11 - Vue aérienne des 4 sphères externes

- Génération et dessin des sphères externes :

Tout comme pour les sphères internes, nous avons créé une fonction permettant de générer et dessiner les sphères externes. Pour ce faire, on parcourt l'intégralité des sphères externes et pour chacune d'elles, on se déplace à ses coordonnées de dessin par le biais d'un **translate**. On rappelle que ces coordonnées sont stockées dans un des tableaux représentant les données sur les sphères externes. Une fois aux bonnes coordonnées, il ne reste plus qu'à les dessiner avec la bonne coloration également stockée dans les structures des sphères externes. Toutes ces opérations sont ainsi réalisées par la fonction nommée "grille_spheresEXT" qui effectue leur génération et leur rendu.

- Détection du regard de l'utilisateur :

A présent que nous avons nos sphères externes, il nous faut offrir la possibilité à l'utilisateur de changer de terrain lorsque celui-ci les regarde. Pour ce faire, nous avons créé une fonction qui parcourt les pixels centraux de l'image et qui détecte si un des pixels à la couleur d'une des sphères externes. Si c'est le cas, alors on incrémente le compteur de regard de l'utilisateur sur cette sphère externe en utilisant une des structures décrites précédemment. On s'assurera évidemment de réinitialiser le compteur de regard de chaque sphère externe si celle-ci n'est pas dans l'axe central de regard de l'utilisateur.

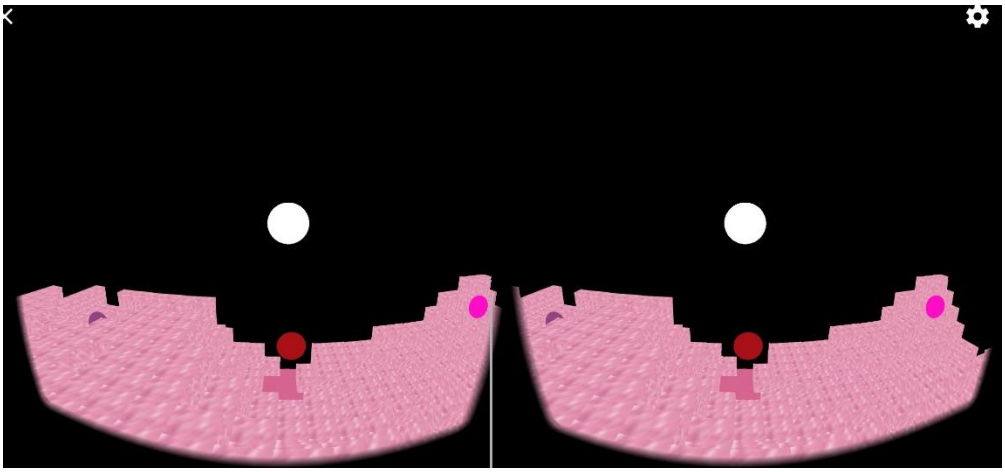


Figure 12 - Utilisateur observant une sphère externe au centre de sa vision (couleur blanche pour le monde Neige)

- Régénération du terrain :

L'objectif principal de ces sphères externes est de permettre à l'utilisateur de pouvoir changer le terrain en les regardant. A présent que nous disposons d'un système de détection du regard de l'utilisateur, voyons comment est effectuée la régénération du terrain. Pour ce faire, nous avons créé une fonction nommée "regenerationRegardSpheresExternes" qui parcourt le tableau "compteur regard" et si une des sphères comptabilise un compteur de regard de 30, correspondant à environ 2 secondes, alors on considère que l'utilisateur a bien regardé cette sphère externe fixement. On connaît alors la sphère qui a été observée et il ne reste plus qu'à réinitialiser tous les compteurs de regards (internes et externes) pour réinitialiser complètement le regard de l'utilisateur. On change également la valeur d'une variable booléenne nommée "reGenerer" à true, pour signifier qu'il faut régénérer le terrain. Enfin, il est essentiel de changer le mode du terrain (nature, lave, neige ou candy) selon la sphère externe qui a été détectée. Ainsi, grâce à cette fonction, un regard prolongé de l'utilisateur sur une sphère externe permet d'autoriser une régénération de terrain qui sera effectuée au prochain **draw**.

Remarque : un affichage console vient faciliter le débogage du programme lors d'une détection de sphère externe accompagnée d'une régénération du terrain.

D. Implémentation globale

Pour présenter l'implémentation globale, nous allons tout d'abord décrire le Pré-Setup, puis le Setup et enfin le contenu du Draw.

- Le Pré-Setup :

Le Pré-Setup constitue l'ensemble des déclarations globales de notre programme. Nous commençons par l'importation des deux bibliothèques traitées précédemment, puis nous déclarons des PShape et des PImage correspondant respectivement à la forme des cubes et leur texture associée. Nous réalisons ceci pour les 4 modes précédemment cités. Ensuite, nous déclarons des constantes correspondantes aux dimensions du terrain, à la taille des cubes, à la hauteur maximale du terrain, une variable **rajout_dessous** permettant de rajouter un certain nombre de cubes en dessous de la surface du terrain, et enfin une variable **flying** qui correspond à la valeur de génération du terrain appliquée au bruit de Perlin.

Pour ce qui suit, nous avons différentes déclarations liées à la génération des sphères internes, des sphères externes et de la génération du terrain.

- Le Setup :

Pour le setup, nous effectuons les affectations de variables et une génération des deux terrains 2D (issu du bruit de Perlin) et 3D aux dimensions nécessaires. Par ailleurs, afin d'obtenir un rendu en Réalité Virtuelle, nous mettons la sortie de l'écran en mode STEREO, puis le terrain est mis en mode nature par défaut, et les coordonnées translate sont initialisées par défaut pour mettre le joueur au centre du terrain au début du programme. Le setup se termine alors sur une initialisation de la sphère actuelle à 4, correspondant à la sphère centrale du terrain.

- Le Draw :

Quant au Draw, afin d'éviter une régénération visible constante du terrain, on met le background à 0 qui joue sur la coloration en noir du fond de la scène. Puis, afin d'effectuer la régénération du terrain lorsque celle-ci est nécessaire, nous utilisons un bloc conditionnel avec un booléen initialisé à true pour que la première génération s'effectue. Dans ce bloc, on réinitialise les deux terrains (le terrain issu du bruit, et le terrain final issu de la conversion en 3D). Puis, afin de changer la

génération aléatoire du terrain, on incrémente le **flying** avec la dimension du terrain. Par la suite, on effectue le calcul du nouveau terrain par bruit de Perlin puis une conversion de ce dernier, en terrain final 3D. La variable booléenne préalablement à "true" prend alors la valeur "false" car le terrain a été régénéré et qu'il n'est plus nécessaire de le modifier pour le moment.

Par la suite, on utilise un **pushMatrix()** et un **popMatrix()** pour encadrer les opérations suivantes et ainsi garantir la sécurité des positions. Les opérations effectuées sont tout d'abord le calcul de la hauteur maximale sur tout le terrain généré ainsi que le calcul de la hauteur du terrain à la position actuelle de l'utilisateur. Par la suite, nous effectuons un rotate sur l'axe X de 90° pour mettre le terrain 3D à "plat" et ainsi permettre à l'utilisateur d'avoir une immersion correcte. Une des phases les plus importantes de ce **Draw** est le déplacement par le **translate**. En effet, on effectue un **translate** aux coordonnées de dessin du terrain grâce aux variables « posTranslateX » et « posTranslateY » et à la hauteur du terrain à la position actuelle de l'utilisateur, calculée précédemment. C'est donc ce **translate** qui permet de dessiner le terrain au bon emplacement dans l'espace par rapport à l'utilisateur et donc de le déplacer si nécessaire. Une fois le **translate** effectué, on peut réaliser le dessin du terrain, des sphères internes et des sphères externes.

Enfin, le **Draw** se conclut sur deux phases essentielles : la première étant la phase des sphères internes durant laquelle on détecte le regard de l'utilisateur sur celles-ci et où on effectue un mouvement de terrain (donc de l'utilisateur par illusion) lorsque cela est nécessaire. La seconde phase étant celle des sphères externes durant laquelle on détecte également le regard de l'utilisateur sur celles-ci et où on effectue la régénération du terrain en fonction d'un mode si cela est nécessaire.

Le **Draw** permet ainsi de réaliser en **continu** le rendu visuel du terrain 3D pour l'utilisateur tout en offrant la possibilité à celui-ci d'effectuer des déplacements et des générations de différents mondes.

V - Difficultés rencontrées

Dans cette partie, nous allons nous intéresser aux difficultés que nous avons rencontrées durant la réalisation de ce projet.

A. L'approche du bruit de Perlin

Comme nous l'avons décrit précédemment, nous avons utilisé le bruit de Perlin pour générer un terrain dynamique 3D uniforme. Cependant, au tout début de ce projet, nous avions eu quelques difficultés de compréhension de ce principe algorithmique fort intéressant. En effet, il nous a fallu effectuer de nombreuses recherches et des phases de tests pour mieux comprendre et prendre en main cette génération uniforme. Après avoir mieux compris son fonctionnement, nous avons pu voir son potentiel et notamment dans le domaine de l'imagerie et du cinéma pour pouvoir par exemple créer de la fumée, des nuages ou encore des flammes.

Figure 13 - Feu réalisé par un bruit de Perlin

B. La désillusion de camera()

Le plus gros problème auquel nous avons dû faire face, a été l'utilisation de la commande camera(). En effet, au début du projet nous avions opté pour faire déplacer l'utilisateur et donc le point de vue utilisateur et non pas le terrain. Pour cela nous avions utilisé les commandes liées à **camera** dans Processing pour déplacer la vision de l'utilisateur sur le terrain. Ce principe fonctionnait malgré sa complexité, mais malheureusement nous avons appris par la suite qu'il était impossible d'utiliser les commandes de **camera** lorsque le rendu serait effectué en mode Android VR. En effet, le rendu Java fonctionnait parfaitement, mais lors du passage en mode Android et

plus particulièrement en mode VR, le rendu était impossible car le mode VR d'Android utilise lui-même les commandes de **camera** pour effectuer les mouvements visuels lorsque des mouvements de têtes sont effectués par l'utilisateur. Nous étions donc face à une impasse, car nous disposions d'un programme fonctionnel en mode Java mais qui ne pouvait être porté en mode Android VR. Bien que cela fût démotivant sur le moment, nous avons su rebondir en réadaptant l'intégralité de notre programme en changeant notre façon de penser. Nous allions alors utiliser les commandes liées à **translate** pour ainsi faire déplacer le terrain sous l'utilisateur et non plus déplacer le point de vue utilisateur. Ainsi, les déplacements du terrain permis par les commandes de **translate**, nous ont permis de donner l'impression à l'utilisateur que celui-ci se déplace sur le terrain. Une fois notre programme réadapté, nous avons pu le porter sur Android VR pour ainsi obtenir le rendu fonctionnel souhaité. Nous avons donc perdu du temps sur cette phase de développement avec **camera** mais nous aurons l'avantage de savoir des à présent que le mode VR d'Android et les commandes de **camera** ne font pas bon ménage.

C. Rendu Immersif et centrage

Une des difficultés dans ce projet fut également son transfert sous le mode Android VR. En effet, l'objectif principal du projet était de pouvoir porter le programme sous Android pour permettre par l'utilisation d'un casque de Réalité Virtuelle, l'obtention d'un rendu immersif. Nous avions durant le cours de Réalité Virtuel de cette année, découvert comment effectuer ce transfert mais malheureusement, nous n'avions pas travaillé sur la détection du centre du regard de l'utilisateur. Cependant, la détection du centre du regard de l'utilisateur est un aspect très important de notre programme, et sans celui-ci, il serait impossible pour l'utilisateur de se déplacer sur le terrain et de changer de monde. Après de nombreuses recherches et grâce à l'aide et aux conseils de M. MESNARD, nous avons choisi de trouver le centre du regard de l'utilisateur en tâtonnant. Ainsi, le centre du regard de l'utilisateur est calibré sur des smartphones ayant une résolution proche de 2400 x 1080 pixels. Bien évidemment, le programme fonctionne pour des smartphones ne disposant pas parfaitement de cette résolution. Ce choix de calibrage manuel a été choisi à la suite des conseils de M. MESNARD car la détection automatique du centre du regard sur les smartphones aurait employé des techniques bien plus complexes et avancées (densité du smartphone, résolution, etc.), sachant que

l'objectif principal était que cela fonctionne sur nos smartphones respectifs pour être en aptitude à réaliser une démonstration. Ainsi, après calibrage manuel, le programme fonctionne parfaitement et a la capacité d'offrir à l'utilisateur une détection du centre de son regard.

VI - Idées d'améliorations

Il est évident que le rendu de ce projet peut être amélioré. C'est pourquoi, vous trouverez dans cette partie, différentes pistes d'améliorations que nous aurions aimé explorer si nous disposions de plus de temps.

A. Les textures du terrain

Pour augmenter l'immersion de l'utilisateur, il pourrait être intéressant de rajouter davantage de textures pour les différents mondes. En effet, il existe pour le moment que deux textures différentes pour chaque monde et leur répartition est peu hétérogène. Un ajout de textures et une répartition plus hétérogène avec une gestion aléatoire mais uniforme de leur placement pourraient ainsi garantir un meilleur réalisme et une meilleure immersion pour l'utilisateur.

Voici un exemple de différentes textures qui pourraient être utilisées :

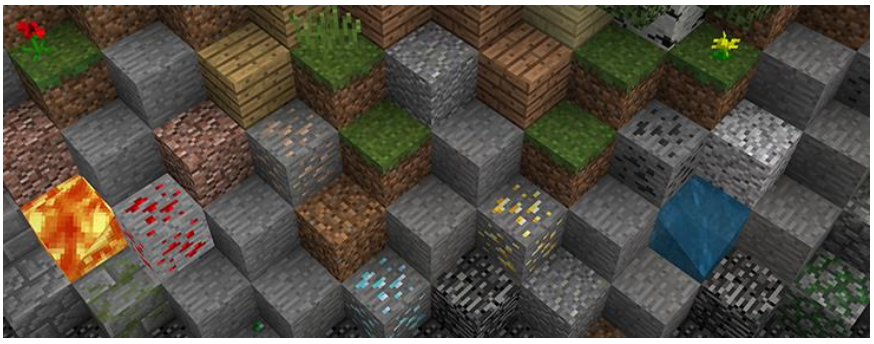


Figure 14 – Exemple de différentes textures

B. Ajout d'éléments de terrain (Arbre, etc.)

Dans l'objectif de garantir une meilleure immersion, il aurait également été intéressant de rajouter différents éléments sur le terrain, que ce soient des arbres pour le monde nature ou encore des sucres

d'orge pour le monde candy. Ces objets pourraient alors être des objets 3D fixe ou encore des objets 3D dynamique.

Remarque : il faudrait tout de même faire attention à ce que l'ajout de ces objets ne dégrade pas la fluidité du rendu.

La liste des éléments à rajouter est sans fin et il faut simplement laisser place à l'imagination, pour rendre encore plus vivants chacun des mondes qui s'ouvrent à l'utilisateur.

C. Ombrages et luminosité

Enfin, la dernière piste d'ajout que nous vous proposons, est d'effectuer une gestion des ombrages et de la luminosité des objets sur le terrain. En effet, après l'ajout d'objets tels que des arbres sur le terrain, il serait intéressant d'ajouter des jeux d'ombres et de lumière pour augmenter le réaliste. L'exemple de l'arbre est simple, mais il est évident que si l'utilisateur peut voir un arbre et l'ombre de celui-ci au sol cela lui garantirait une meilleure expérience immersive. Ainsi, la gestion des ombrages et de la luminosité sur des objets et plus généralement sur l'intégralité du terrain pourrait être une piste d'amélioration de ce rendu, pour encore une fois garantir une meilleure immersion pour l'utilisateur.

Figure 15 - Shaders Minecraft

Conclusion

Pour conclure, ce projet fut très intéressant et nous sommes ravis d'avoir pu le réaliser. Nous remercions M. MESNARD pour cette proposition, qui nous a permis de mieux prendre en main le logiciel Processing mais également de réaliser un monde entièrement virtuel qui peut être exploré par l'intermédiaire d'un casque virtuel.

Nous garderons de ce projet une opportunité de découvrir concrètement la Réalité Virtuelle. Ce fut ainsi, une expérience enrichissante et c'est avec impatience que nous attendons notre prochain projet qui nous permettra, on l'espère, de découvrir une nouvelle facette du domaine de l'Informatique.

Table des figures :