Une île virtuelle en 3D avec Delphi et Firemonkey

Par Grégory Bersegeay
http://www.gbesoft.fr

Nantes le 23 mai 2019

SOMMAIRE

- Création d'un décor extérieur
- Déplacements et orientation
- Gestion des collisions
- Bonus
- Questions/Réponses

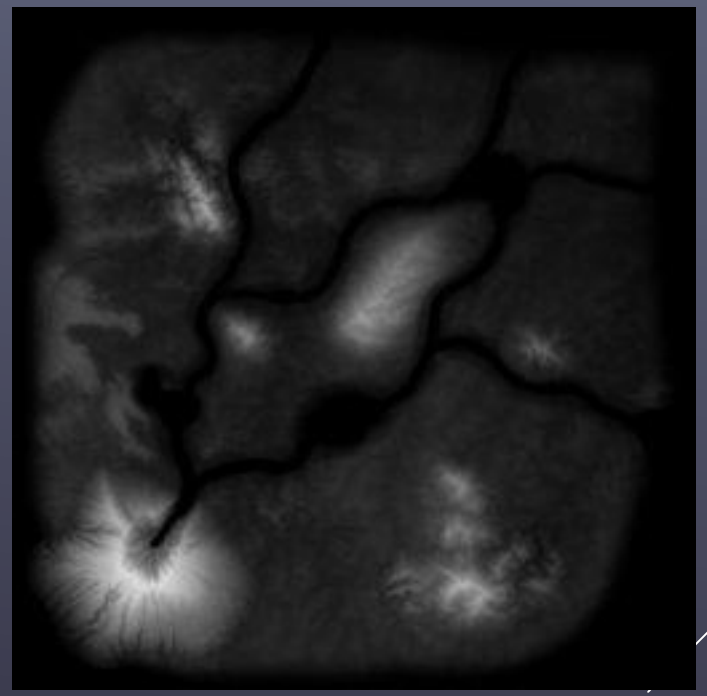
CRÉATION DU DÉCOR (1/4)

Nous allons utiliser la technique du champ de hauteur (heightmap) pour générer le sol du décor.

Cette technique se base sur une image dont la couleur de chaque pixel représentera l'altitude du point concerné.

Voici l'image *heightmap* utilisée pour la version finale de la petite démonstration.

Sur cette image plus le pixel est clair, plus l'altitude du point correspondant sera élevée.



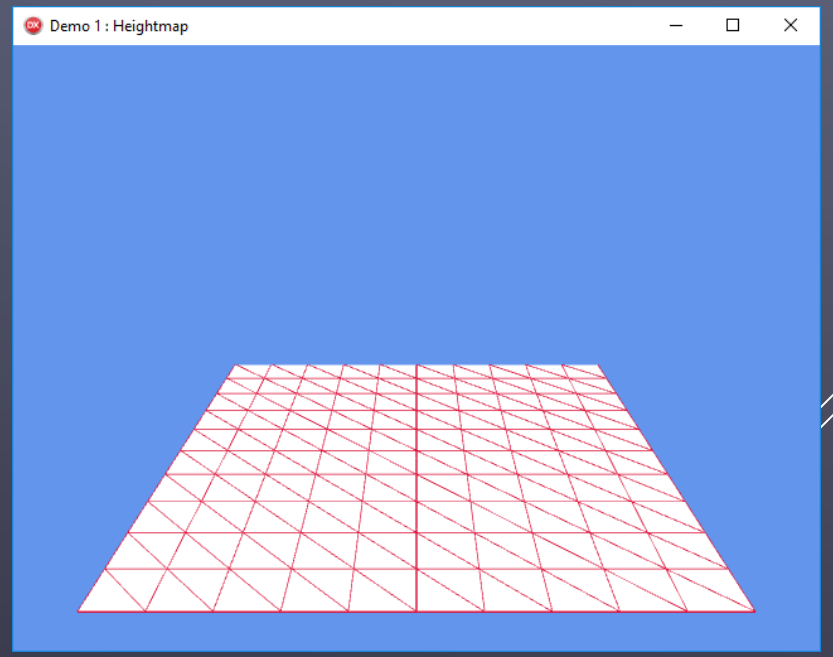
CRÉATION DU DÉCOR (2/4)

Pour créer le décor, nous allons utiliser un objet de type **TMesh**. Cet objet permet de manipuler un objet 3D complexe, c'est-à-dire un ensemble de polygones (des triangles) liés les uns aux autres (dans un maillage ou mesh en anglais). C'est la propriété **Data** de cet objet qui contient toutes les informations de l'objet 3D (en particulier les coordonnées de tous ses sommets).

Il serait possible de définir dynamiquement les informations de chaque maille mais, par facilité, nous allons créer notre **TMesh** à partir d'un **TPlane**.

Le **TPlane** est un plan en 3D qu'il est possible de subdiviser en largeur et hauteur.

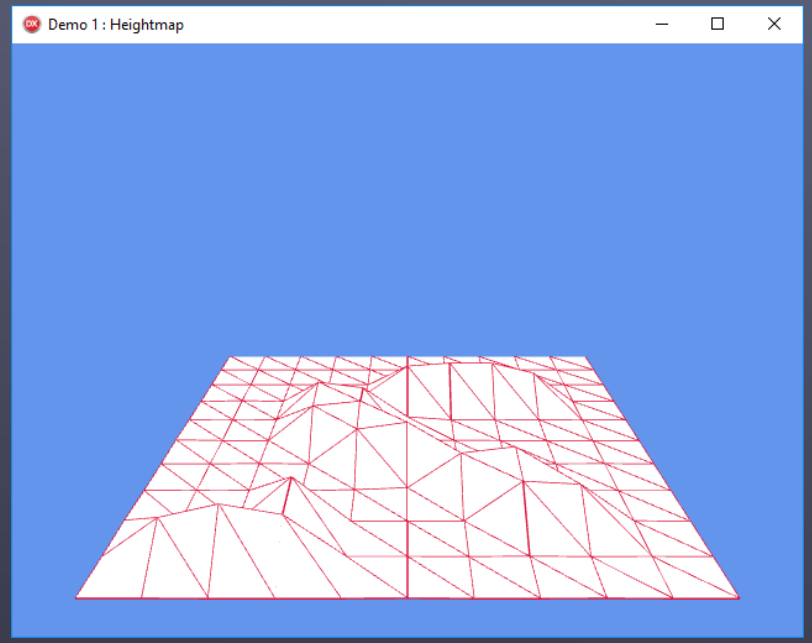
L'image ci contre est un **TMesh** créé à partir d'un **TPlane** subdivisé en 10 zones en largeur et 10 zones en hauteur.



CRÉATION DU DÉCOR (3/4)

Pour créer notre décor, nous allons :

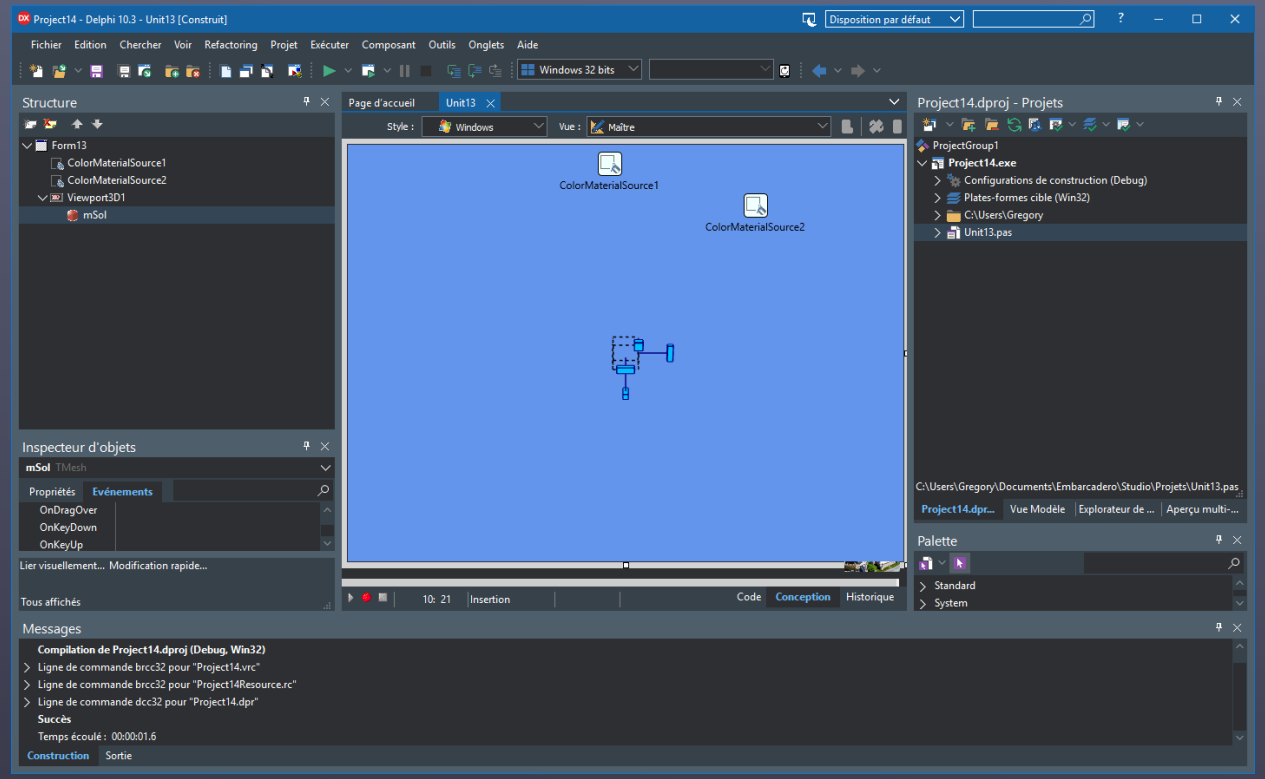
- subdiviser le **TPlane** en fonction de la taille de l'image *heightmap*;
- parcourir l'image *heightmap* pixel par pixel afin de calculer la hauteur du sommet correspondant en fonction de la couleur du pixel;



A noter : en floutant l'image *heightmap*, le relief semble plus doux, plus érodé.

CRÉATION DU DÉCOR (4/4)

Passons à la pratique.



Nantes le 23 mai 2019

DÉPLACEMENTS ET ORIENTATION (1/9)

Nous venons de créer un décor en 3D. Nous allons maintenant permettre à l'utilisateur de s'y déplacer en toute liberté.

Pour gérer le déplacement, nous avons besoin de deux éléments : la vitesse et l'orientation.

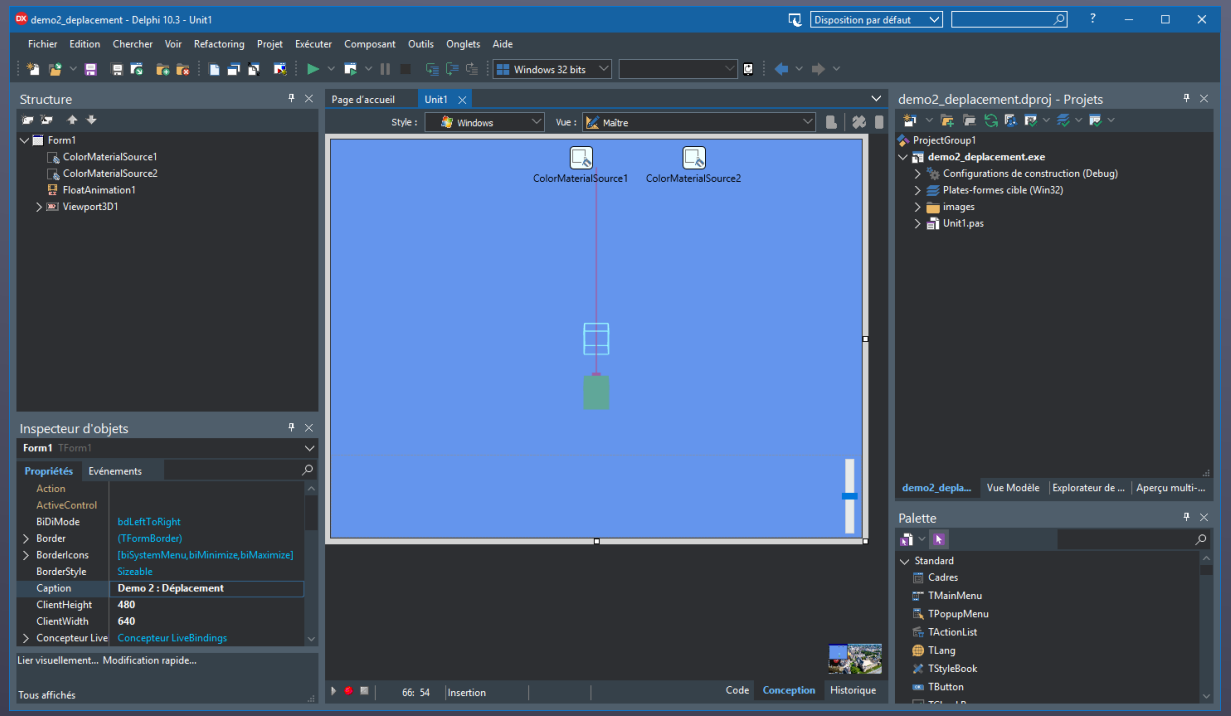
La vitesse sera une simple valeur numérique qui provoquera :

- un déplacement vers l'avant lorsqu'elle sera positive;
- un déplacement vers l'arrière lorsqu'elle sera négative;
- l'arrêt lorsqu'elle sera nulle.

Nous utiliserons un composant de type **TTrackBar** pour gérer cette vitesse.

DÉPLACEMENTS ET ORIENTATION (2/9)

Passons à la pratique.

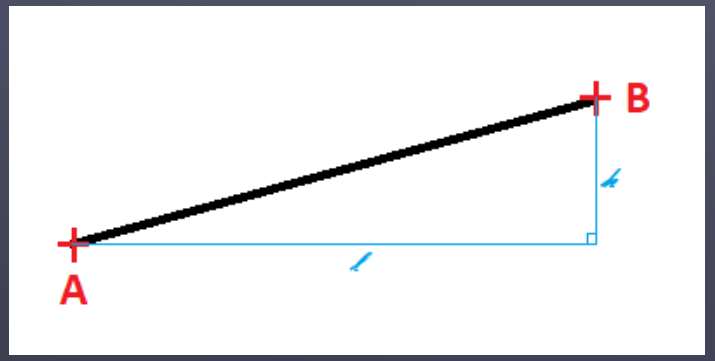


Nantes le 23 mai 2019

DÉPLACEMENTS ET ORIENTATION (3/9)

Nous avançons et nous reculons. C'est bien mais ce n'est pas suffisant pour permettre à l'utilisateur de se déplacer librement. Nous allons donc gérer maintenant l'orientation.

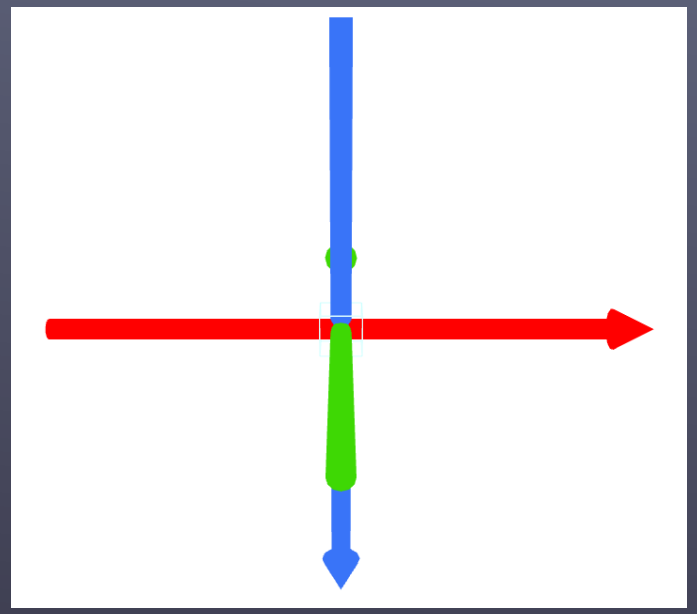
Pour cela, nous allons permettre à l'utilisateur de modifier son orientation à l'aide de la souris.



Imaginons que l'utilisateur clique sur l'écran au point **A**, il déplace la souris tout en maintenant le bouton enfoncé et relâche le bouton au point **B**.

Nous en déterminons **l** la distance entre les points **A** et **B** sur l'axe **X**, et **h** la distance entre **A** et **B** sur l'axe **Y**.

DÉPLACEMENTS ET ORIENTATION (4/9)

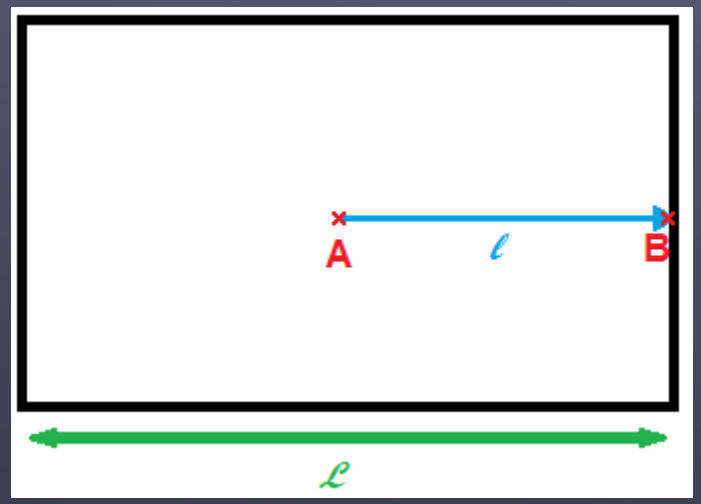


Nous utiliserons les valeurs **l** et **h** pour orienter le point de vue de l'utilisateur de la manière suivante :

- **l** sera utilisée pour déterminer la rotation à effectuer du point de vue de l'utilisateur autour de l'axe Y (orientation droite/gauche) ;
- **h** sera utilisée pour déterminer la rotation à effectuer du point de vue de l'utilisateur autour de l'axe X (orientation haut/bas).

DÉPLACEMENTS ET ORIENTATION (5/9)

Nous allons avoir besoin de gérer une sensibilité. La règle que nous allons utiliser est la suivante : si l'utilisateur clique au milieu de la fenêtre de notre application, puis il déplace la souris horizontalement jusqu'au bord droit de la fenêtre, nous devrons effectuer une rotation de l'orientation de 90°.



Dans l'exemple ci contre, nous pouvons calculer la distance entre les points **A** et **B** sur l'axe **X** et nous connaissons la largeur **L** de la fenêtre.

La rotation à effectuer autour de l'axe **Y** sera donc :

angleY = l x (180 / L)

Un déplacement de longueur **L** sur l'axe **X** équivaut à faire un demi tour.

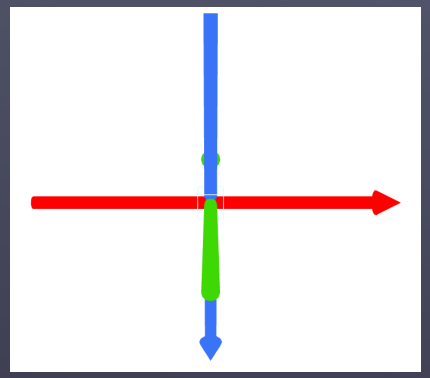
Nous aurons un rapport similaire entre le déplacement en hauteur **h** et l'angle de rotation à appliquer sur l'axe **X : angleX = h * (180 / H)**

DÉPLACEMENTS ET ORIENTATION (6/9)

L'orientation sera donc décomposée en deux temps avec un petit calcul supplémentaire en fonction de la sensibilité. Cette décomposition du mouvement va nécessité une petite astuce.

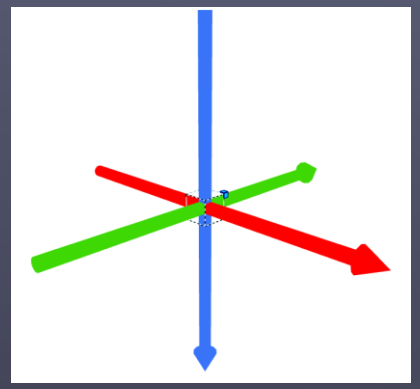
En effet, voici notre repère :

X
Y
Z



Appliquons lui la première composante de l'orientation, une rotation de 45° autour de l'axe **Y** :

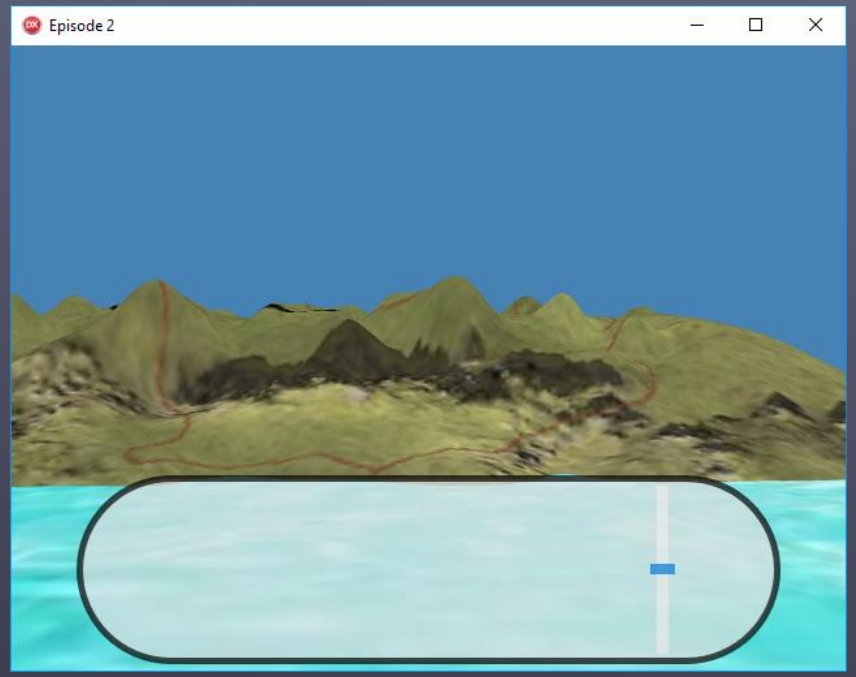
X
Y
Z

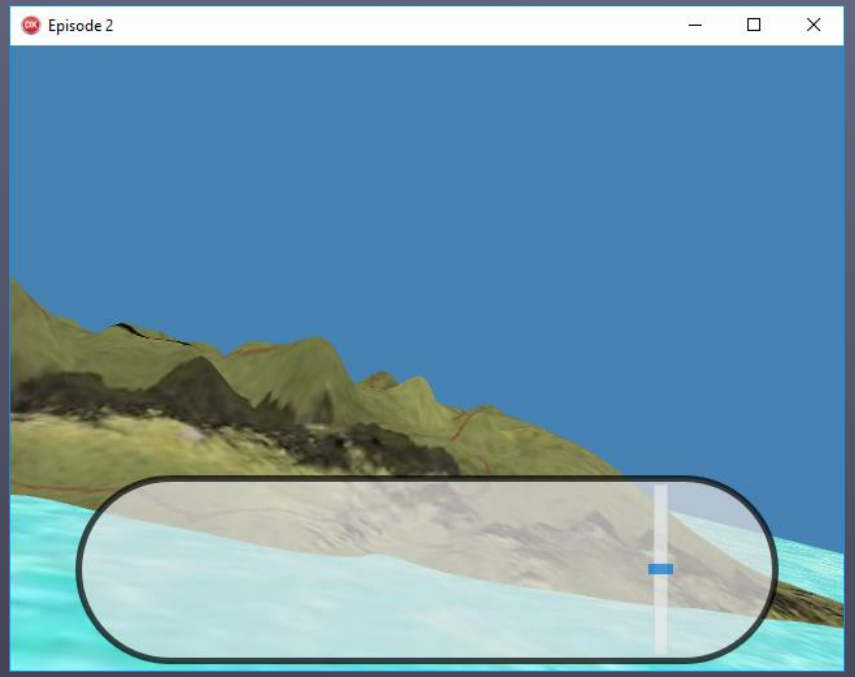


Nous constatons que l'axe **X** a tourné : la valeur de la rotation à appliquer sur l'axe **X** n'est plus la valeur calculée précédemment.

DÉPLACEMENTS ET ORIENTATION (7/9)

Voici ce que cela donnerait dans un cas concret :





Nous aurions l'impression de pencher la tête... Ce n'est pas l'effet recherché !

DÉPLACEMENTS ET ORIENTATION (8/9)

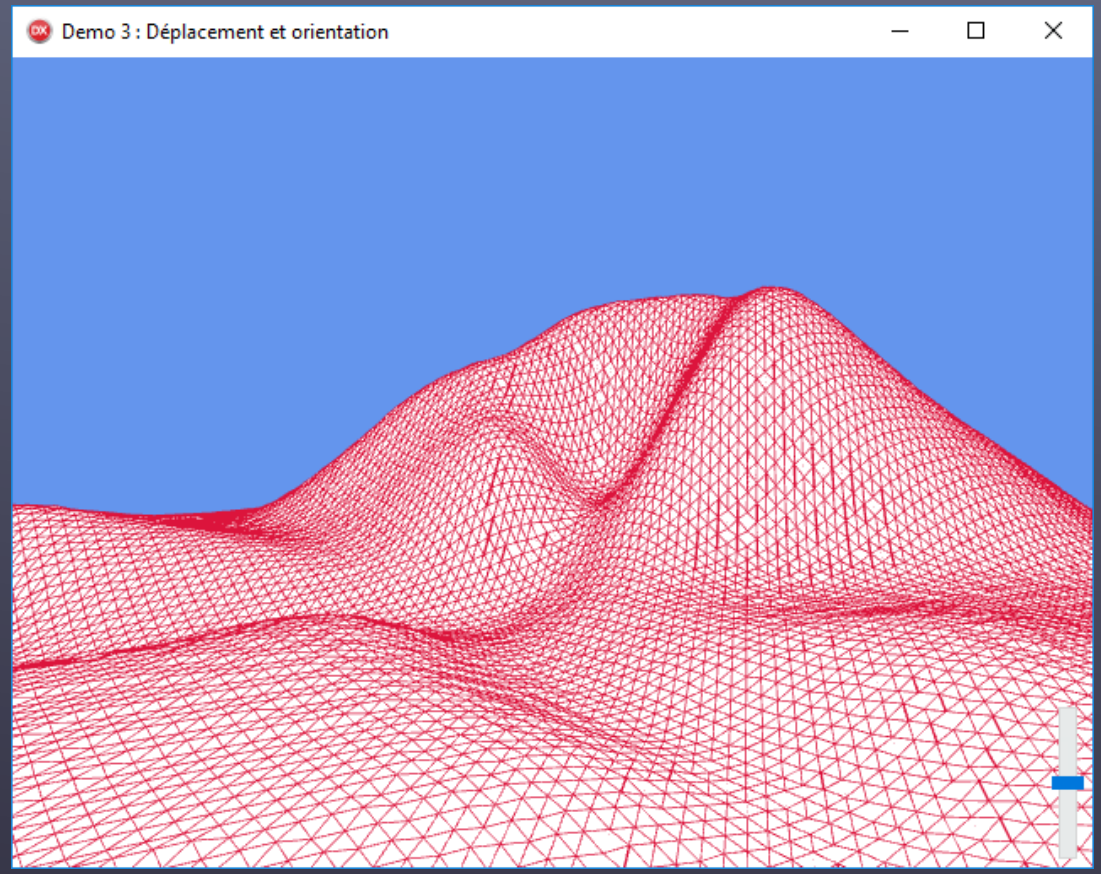
La petite astuce évoquée va permettre de traiter ce point sans entrer dans des calculs complexes.

Elle consiste à placer deux **TDummy**, l'un en tant qu'enfant de l'autre. Nous appliquerons la rotation de valeur **l** autour de l'axe Y du **TDummy** parent et la rotation de valeur **h** autour de l'axe X du **TDummy** enfant.

Pour bénéficier de la composition des deux rotations, la caméra sera placée en tant qu'enfant du second **TDummy**.

DÉPLACEMENTS ET ORIENTATION (9/9)

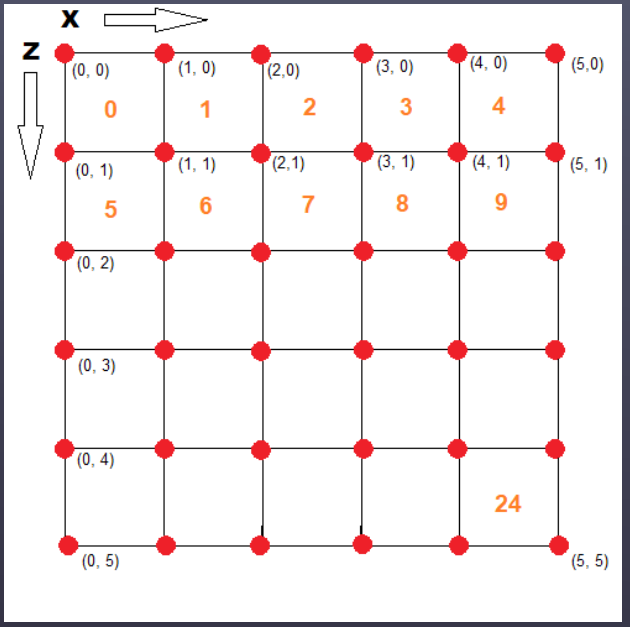
Passons à la pratique.



GESTION DES COLLISIONS (1/4)

Nous avons notre décor et nous pouvons nous y déplacer librement. Tellement librement qu'il est possible de passer à travers le décor…Remédions à cela !

Pour le moment, nous n'avons pour décor que le **TMesh** représentant le sol. Nous allons donc voir comment placer la caméra afin que l'utilisateur ait l'impression de suivre les aspérités du sol. Il va donc nous falloir connaître la hauteur d'un point quelconque sur notre **TMesh**.



Ce schéma représente notre maillage vu de dessus. L'objet **TMesh** contient une collection de mailles dans sa propriété **Data**. Nous avons accès aux coordonnées des sommets de chaque maille grâce à la propriété **Data.VertexBuffer.Vertices**.

Exemple : pour obtenir la hauteur du sommet aux coordonnées x et z, nous ferons :
mesh.data.VertexBuffer.Vertices[floor(x) + (floor(z) * 5)].Z

Floor(x) renvoie le plus grand entier inférieur ou égal à x.

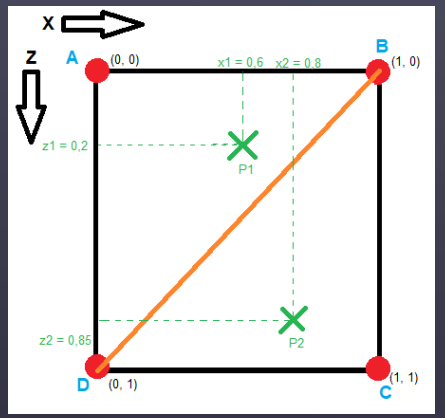
DX barnsten

GESTION DES COLLISIONS (2/4)

Nous savons maintenant récupérer la hauteur d'un sommet proche de la position du joueur. Le problème qui se pose à présent est qu'une maille est grande et que le joueur peut être positionné n'importe où sur la maille. Si nous nous contentons des hauteurs des sommets avoisinant la position du joueur, le déplacement ne sera pas réaliste et se fera par à-coups donnant l'impression de monter ou descendre un escalier.

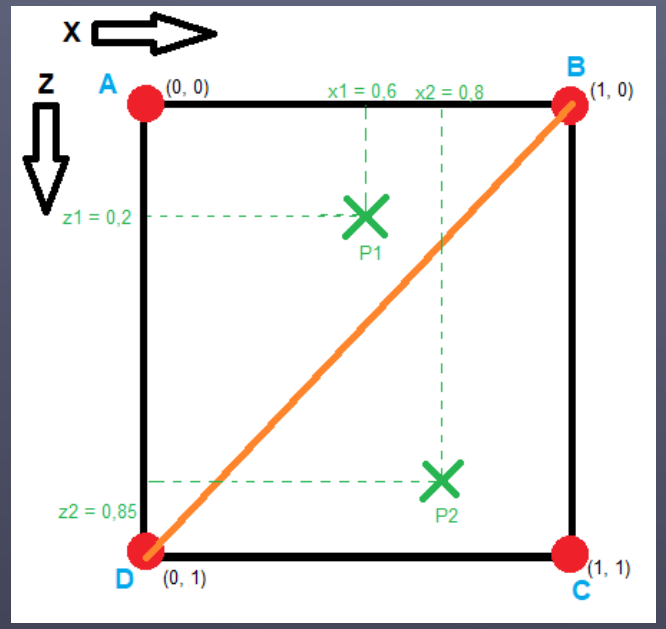
Nous allons devoir calculer des valeurs de Y intermédiaires en fonction des valeurs de Y de chaque sommet. C'est ce qui s'appelle une interpolation.

Zoomons sur une des mailles du schéma précédent :



Jusqu'à présent, j'ai représenté par un quadrillage les subdivisions du **TPlane** ayant servi à modeler le **TMesh**. En réalité, les moteurs graphiques n'utilisent pas des carrés mais des triangles.

GESTION DES COLLISIONS (3/4)



Les sommets A, B, C et D du carré sont les quatre points pour lesquels nous connaissons exactement la hauteur Y via **mSol.Data.VertextBuffer.Vertices**.

En traçant la diagonale orange telle qu'elle est indiquée sur le schéma, elle découpe le carré en deux triangles ABD et BCD.

Nous devons donc déterminer dans quel triangle se trouve le joueur.

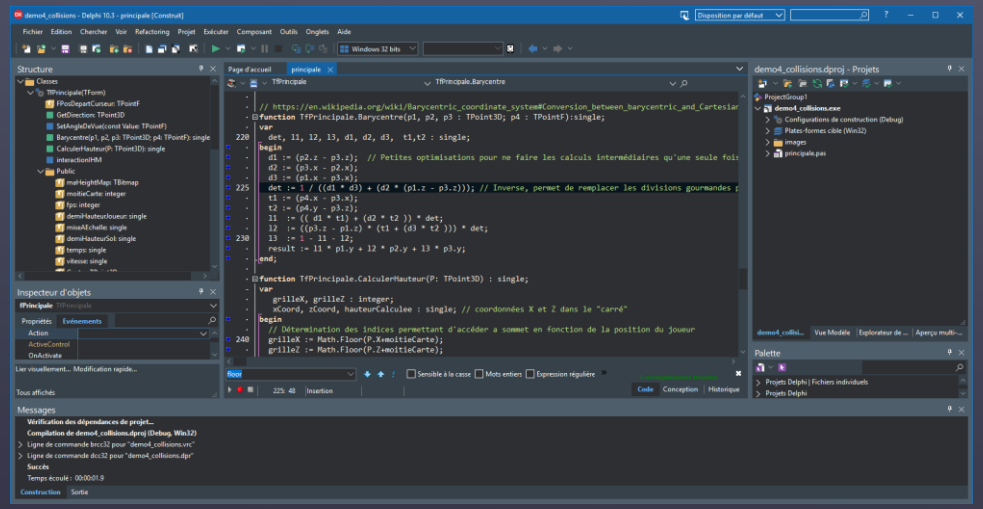
Pour ce faire, nous remarquons que les points situés sur la diagonale orange ont des coordonnées X et Z liées suivant la règle : X = 1 - Z.

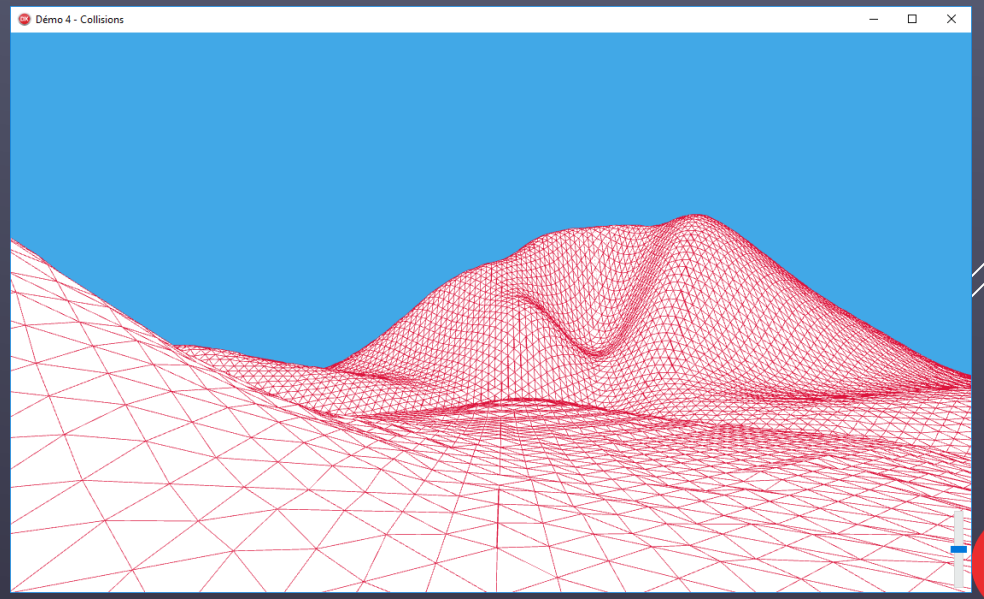
Nous pouvons en déduire que si X < 1 – Z, alors le point est dans le triangle ABD sinon, il est dans le triangle BCD.

GESTION DES COLLISIONS (4/4)

Sachant dans quel « triangle » est positionné le joueur, nous allons pouvoir calculer une hauteur en fonction des hauteurs des trois sommets du triangle. Pour cela, nous allons utiliser la méthode des barycentres (décrite ici : https://en.wikipedia.org/wiki/Barycentric_coordinate_system#Barycentric_coordinates_on_triangles).

Je ne m'étendrai pas sur les explications mathématiques, passons tout de suite à la pratique ☺



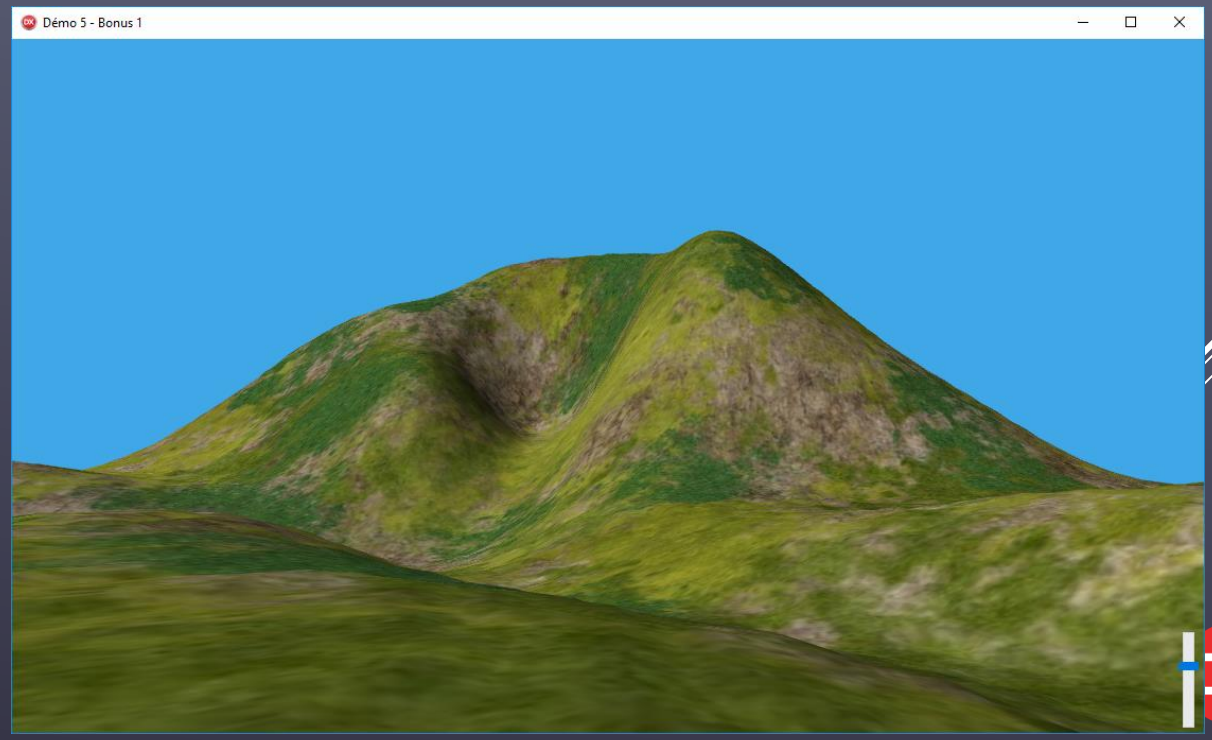


BONUS (1/5)

Nous avons vu quelques éléments de base pour la création de décors en 3D.
Je vous avais promis la création d'une île en 3D et force est de constater que nous en sommes encore loin…

Nous allons donc ajouter quelques éléments qui amélioreront le rendu et le réalisme.

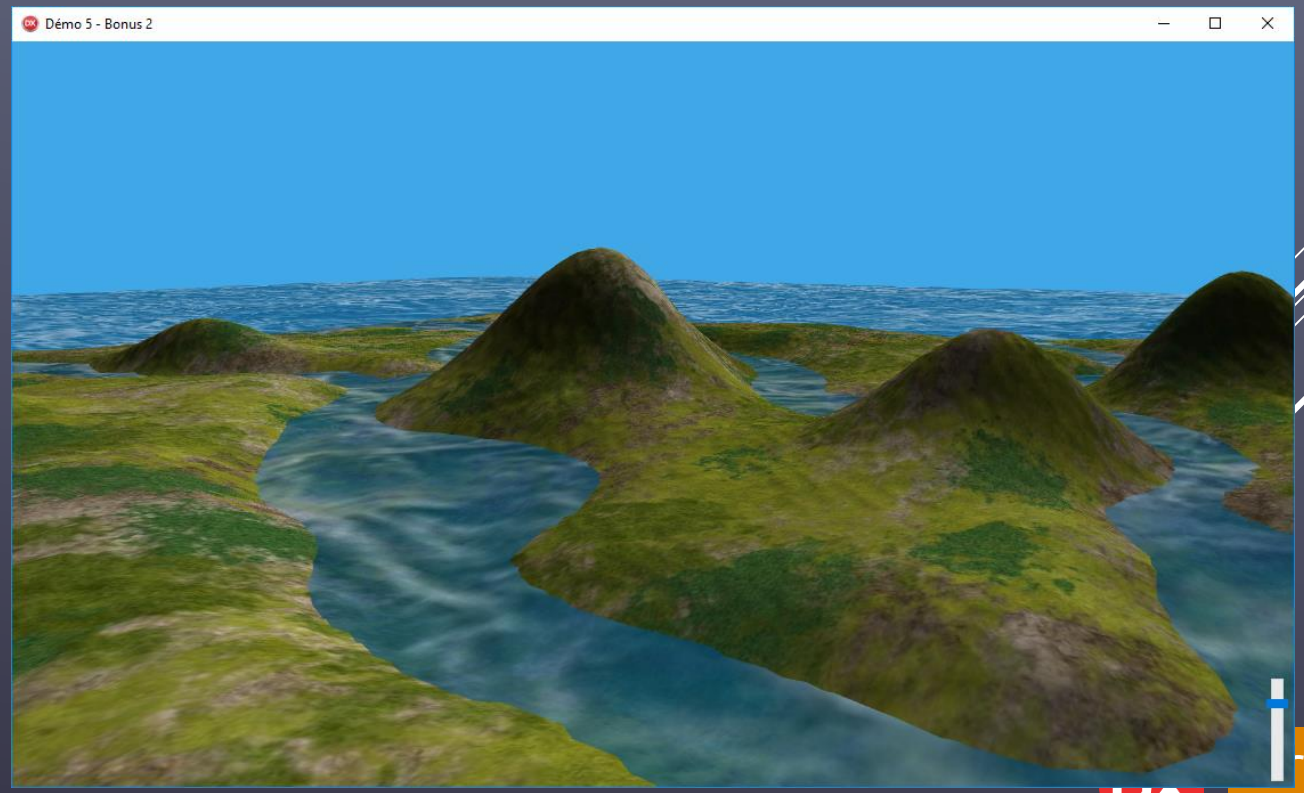
Tout d'abord, ajoutons une source lumineuse et une texture sur le **TMesh** :



BONUS (2/5)

Une île est par définition entourée d'eau. Ajoutons un nouveau **TPlane** et associons lui une texture. Il symbolisera la mer. Nous allons le dimensionner de telle sorte qu'il soit bien plus grand que notre **TMesh**. Nous le placerons à une certaine hauteur afin que les mailles basses du **TMesh** soient sous le niveau de la mer.

Il est même possible de le subdiviser afin de jouer dynamiquement sur les hauteurs des sommets des mailles et ainsi simuler des vagues :



BONUS (3/5)

Augmentons encore un peu plus le réalisme gérant des nuages dans le ciel.

Cette fois ci nous allons ajouter dynamiquement n **TPlane** qui :

- seront de taille aléatoire;
- seront orientés aléatoirement;
- seront texturés aléatoirement;
- seront placés aléatoirement.

Ils se déplaceront dans la même direction mais à des altitudes et vitesses différentes.

BONUS (4/5)

Il est également possible d'importer des objets 3D provenant de logiciels de modélisation. Delphi gère nativement les formats ASE, DAE et OBJ.

Ajoutons quelques palmiers sur notre île. Maintenant, il faut gérer les collisions du joueur avec les palmiers.

Dans cette démonstration, nous allons utiliser la technique des « bounding box » : nous allons détecter une collision entre la position du joueur et le rectangle 3D imaginaire englobant le palmier.
Cette technique n'est pas précise, mais elle est simple à mettre en œuvre et suffisante pour cet exemple.

BONUS (5/5)

Enfin, dernier bonus important, tout ce que nous venons de voir fonctionne tel quel sans modification de code sur Windows, Mac OS X, Android et IOS (et avec la l'édition Community de Delphi qui est gratuite) :

Avez-vous des questions ?

Nantes le 23 mai 2019

Merci ☺

Grégory Bersegeay

Site web : http://www.gbesoft.fr
Mail : gregory.bersegeay@gbesoft.fr
GitHub : https://github.com/gbegreg
Les sources des démos présentées sont disponibles sous https://github.com/gbegreg/Ile3D
Mon pseudo sur Developpez.com : gbegreg

Nantes le 23 mai 2019