Une course de voiture en 3D avec Delphi et Firemonkey

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

SOMMAIRE

- Création d'un décor extérieur
- Création d'un véhicule
- Déplacements et orientation
- Suivre les aspérités du terrain
- Détection des collisions avec les obstacles
- Bonus
- Questions/Réponses

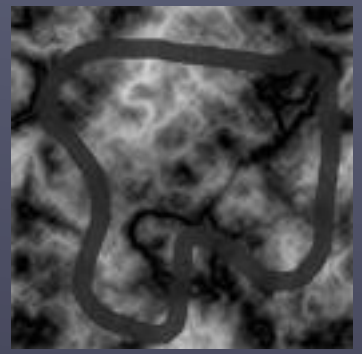
CRÉATION DU DÉCOR

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 FMX Race.

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



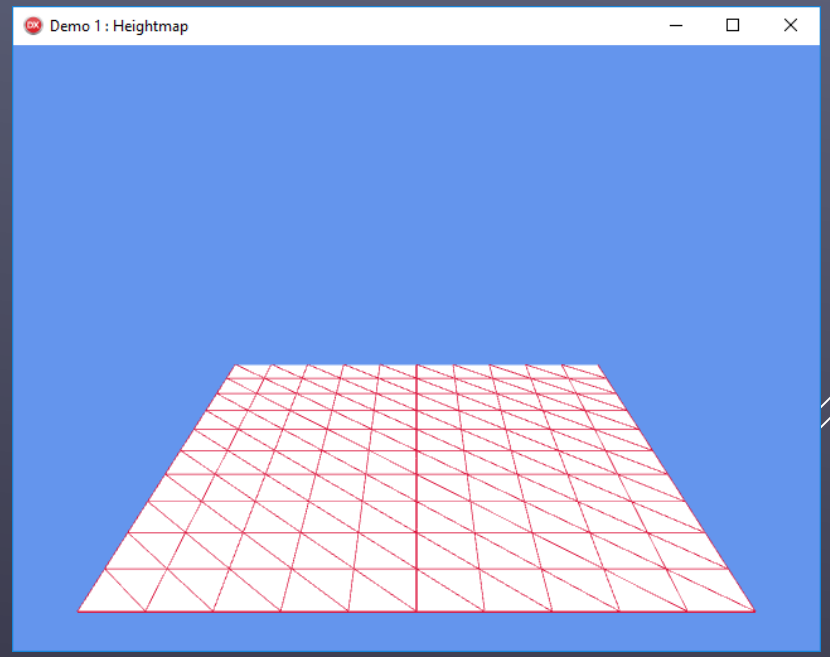
CRÉATION DU DÉCOR

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.

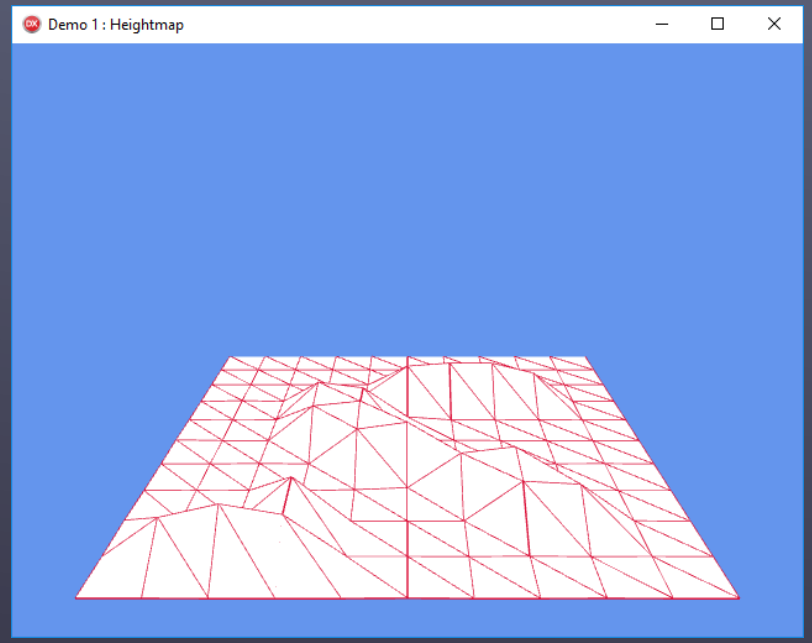
L'image ci contre est un **TMesh** créé à partir d'un **TPlane** subdivisé en 10 zones en largeur et 10 zones en hauteur.



CRÉATION DU DÉCOR

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;

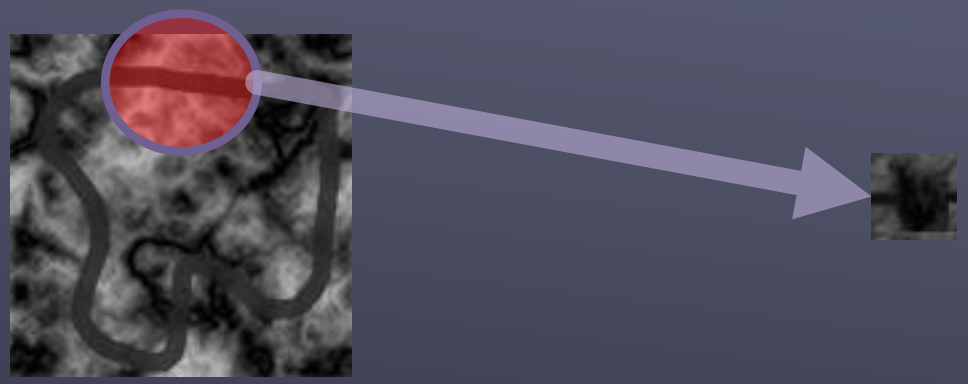


A noter : en floutant l'image *heightmap*, le relief semble plus doux, plus érodé.

CRÉATION DU DÉCOR

Dans le projet FMX Race, nous utiliserons deux **TMesh** basés sur la technique du heightmap.

En effet, nous allons séparer le sol en deux : le premier **TMesh** (nommé **mSol**) symbolisera le sol sur lequel la voiture va rouler, le second **TMesh** (nommé **mMontagne**) correspondra à la Montagne.



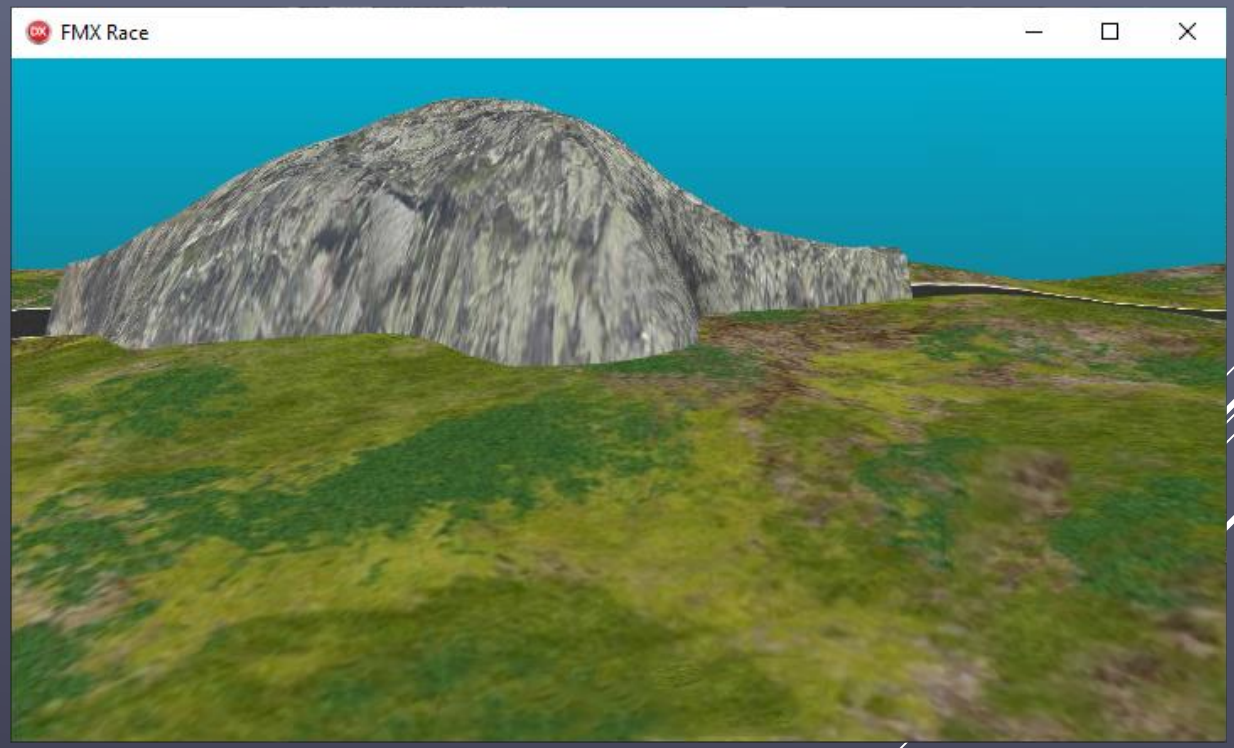
mSol

mMontagne

CRÉATION DU DÉCOR

Les deux **TMesh** seront positionnés de telle sorte qu'une partie du **mMontagne** transperce **mSol**.

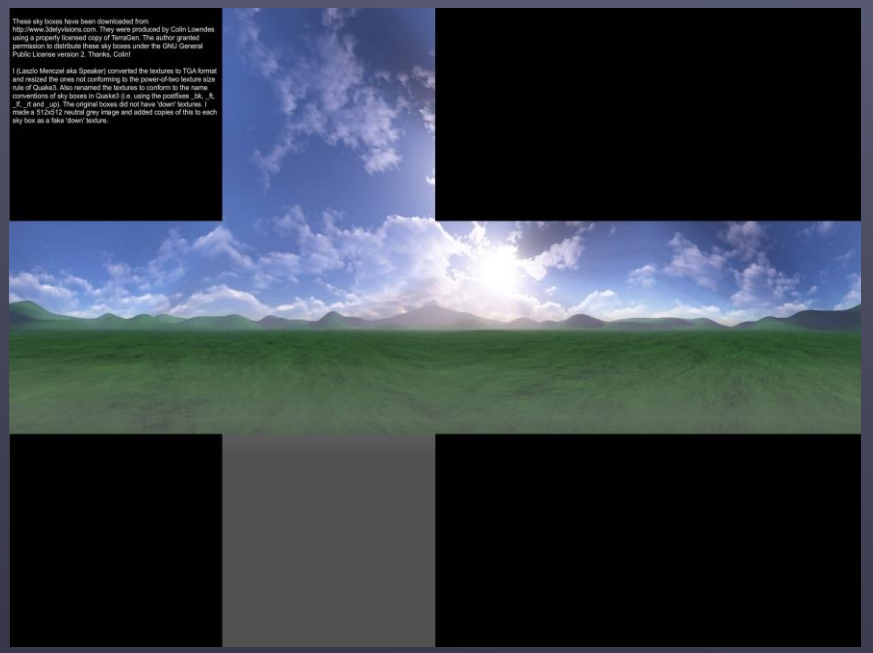
Ici, **mSol** est texturé avec la texture du sol et **mMontagne** avec la texture grise.



CRÉATION DU DÉCOR

Nous allons ajouter un horizon à notre scène. Pour ce faire, nous allons utiliser un **cubemap**. Le cubemap est tout simplement un cube suffisamment grand. Nous devrons en texturer chacune de ses 6 faces de sorte que l'utilisateur qui sera placé à l'intérieur ait l'impression d'être plongé dans le décor qu'on lui propose.

Voici la texture que nous allons utiliser pour FMX Race :



L'image provient du site http://www.3delyvisions.com

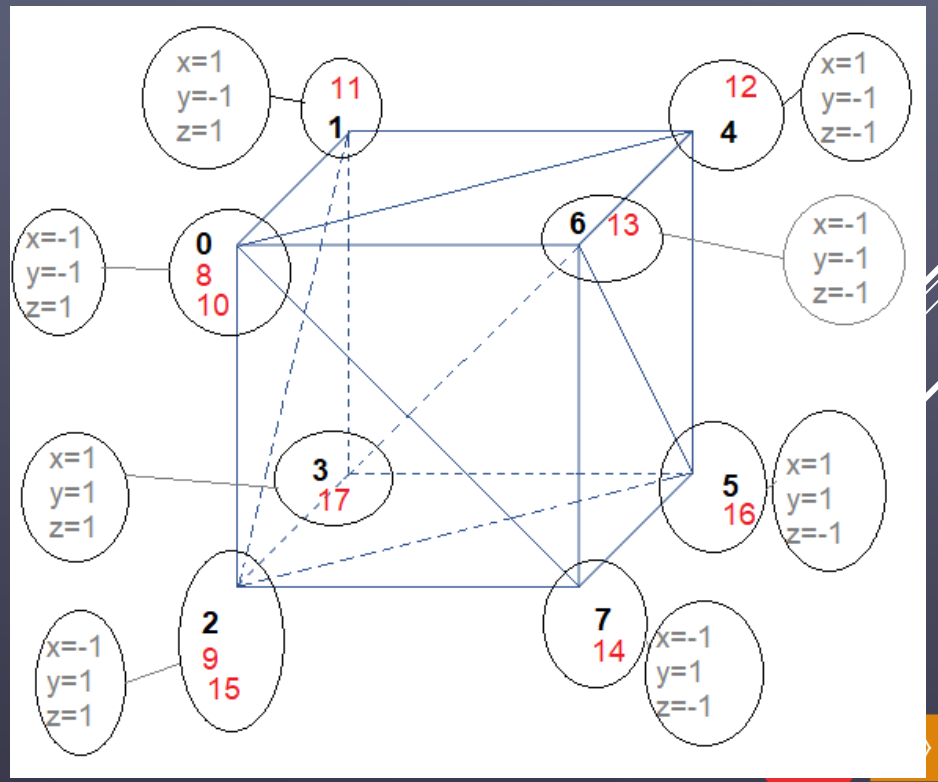
CRÉATION DU DÉCOR

Par défaut, lorsqu'on applique une texture à un TCube, l'image s'applique entièrement sur chacune des faces du cube. Ce n'est pas ce qu'on souhaite ici.

Nous allons donc associer à chaque sommet du cube les coordonnées de la texture que l'on souhaite appliquer à ce sommet.

Pour définir un cube, il nous faut définir 8 sommets. Le problème est que nous ne pouvons affecter à un sommet qu'une seule coordonnée de la texture.

Il va falloir dupliquer des sommets pour pouvoir appliquer la texture comme on le souhaite. Dans notre cas, nous aurons besoin de 18 sommets.



CRÉATION DU DÉCOR

Nous allons utiliser les propriétés "texte" **Data.Points**, **Data.TexCoordinates** et **Data.TriangleIndices** du **TMesh** afin de pouvoir appliquer la texture comme nous le souhaitons à notre cube (de 18 sommets ☺).

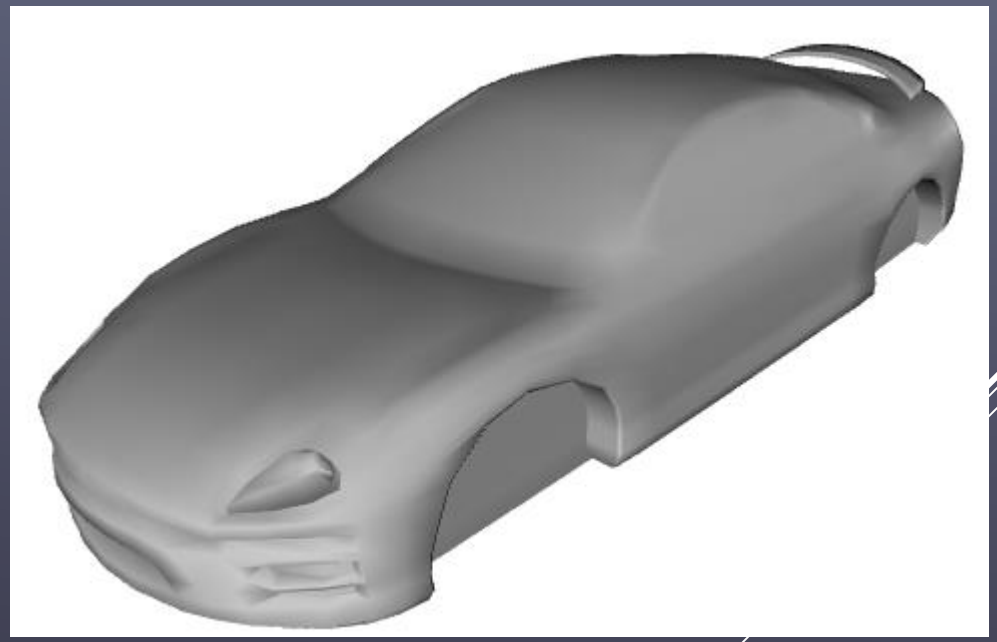
```
// 18 points pour pouvoir appliquer la texture correctement (8 points sufisent pour le cube, mais on ne peut ensuite associer
// qu'un point de la texture à un sommet, alors on duplique les sommets nécessaires pour pouvoir appliquer la texture correctem
self.Data.Points :=
    '-1 -1 1,  1 -1 1,  -1 1 1,  1 1 1, 1 -1 -1, 1 1 -1, -1 -1 -1, -1 1 -1, -1 -1 1, -1 1 1,'+   // faces Gauche, Face, Droit,
    '-1 -1 1, 1 -1 1, 1 -1 -1, -1 -1 -1, -1 1 -1, -1 1 1, 1 1 -1, 1 1 1';                         // faces Haut et Bas
// Positionnement de la texture à chaque points
self.Data.TexCoordinates :=
    '0.0 0.34, 0.25 0.34, 0.0 0.66, 0.25 0.66, 0.5 0.34, 0.5 0.66, 0.75 0.34, 0.75 0.66, 1 0.34, 1 0.66,'+
    ' 0.25 0.0, 0.25 0.34, 0.5 0.34, 0.5 0.0, 0.5 1, 0.25 1, 0.5 0.66, 0.25 0.66';
// Création et indexation des triangles en fonction du besoin
self.Data.TriangleIndices := '0 1 2 ,2 1 3 ,1 4 3, 3 4 5, 4 6 5, 5 6 7, 6 8 7, 7 8 9, 10 11 12, 12 10 13, 14 15 16, 16 15 17';
```

J'ai réalisé un composant **TGBECubemap** qui hérite du **TMesh** et qui permet de générer un cubemap. C'est ce qui sera utilisé dans le projet *FMX Race*.

CRÉATION D'UN VÉHICULE

Etant bien incapable de modéliser une voiture, j'ai récupéré sur internet un modèle gratuit : https://www.turbosquid.com/3d-models/free-max-model-2003-mitsubishi-eclipse/577425

Pourquoi ce modèle ?
- il est gratuit;
- il est fourni avec plusieurs textures;
- composé de peu de polygones (moins de 8000).

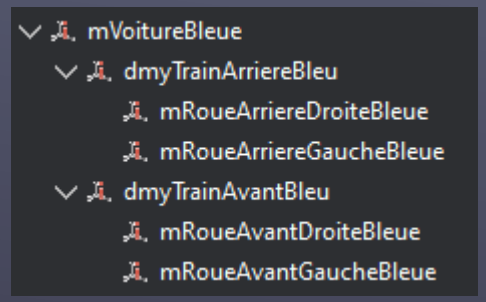


J'ai modifié le modèle d'origine pour le simplifier et surtout dissocier les roues afin de pouvoir les animer ensuite indépendamment du reste de l'objet.

J'ai supprimé les roues d'origine mais également les rétroviseurs et les essuis glaces. J'ai trouvé un autre modèle 3D de roue.

CRÉATION D'UN VÉHICULE

Sous Delphi, j'ai constitué le véhicule via le designer directement avec la structure suivante :



mVoitureBleue est le **TModel3D** dans lequel le modèle de la voiture est chargé.

Il contient deux **TDummy** : **dmyTrainArriereBleu** et **dmyTrainAvantBleu**. Comme leurs noms l'indiquent, il s'agit des trains roulant.

Chacun des trains contient deux **TModel3D** qui représentent les roues.

Les autres voitures seront clonées à partir de **mVoitureBleue** et les composants enfants seront renommés dynamiquement dans le code.

DÉPLACEMENTS ET ORIENTATION

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

Notre démonstration doit être utilisable sur ordinateur mais aussi sur smartphone et tablette.
J'ai donc choisi de simuler un joystick virtuel pour permettre à l'utilisateur de déplacer la voiture.

Voici le joystick virtuel :

Le cercle central sera cliquable et l'utilisateur pourra le déplacer dans toutes les directions.

La composante verticale de ce déplacement correspondra à l'accélération/décélération et influera donc la vitesse.
La composante horizontale de ce déplacement correspondra à la modification de l'orientation.

Lorsque l'utilisateur lâchera le joystick, celui-ci se repositionnera au centre et la vitesse reviendra progressivement à zéro.

DX barnsten

DÉPLACEMENTS ET ORIENTATION

Grace au joystick virtuel, nous allons pouvoir déplacer librement la voiture du joueur.

Pour la gestion de la vitesse, c'est simple : il suffira de faire avancer ou reculer la voiture.

Pour la direction en revanche, il faudra tourner la voiture, bien sûr, mais également qu'elle se dirige dans la bonne direction.
Pour ce faire, nous utiliserons la direction donnée entre la caméra 1 et la voiture du joueur (fonction **GetDirection**).

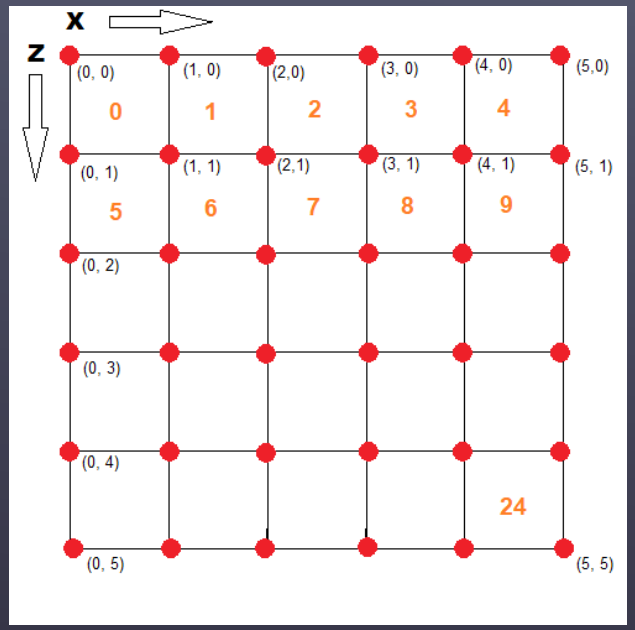
```
//-------- Gestion de la direction ------------------------------------------------------
function TfPrincipale.GetDirection: TPoint3D;
begin
  result := Point3D(1,0,1) * (TPoint3D(Cameral.AbsolutePosition) - TPoint3D(dmyPositionJoueur.AbsolutePosition));
end;
```

DX barnsten

SUIVRE LES ASPÉRITÉS DU TERRAIN

Nous avons notre décor et nous pouvons nous y déplacer librement. Il va maintenant falloir que la voiture suive le relief du sol.

Pour cela, nous devons connaître la hauteur d'un point quelconque sur notre **TMesh mSol**.



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.

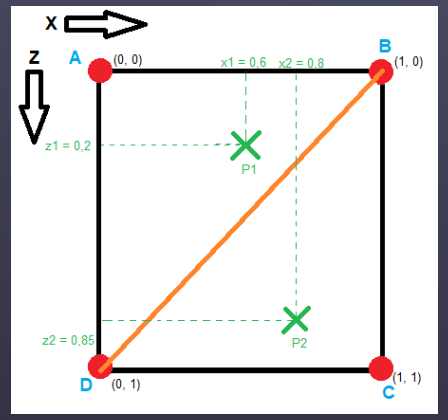
DX barnsten

SUIVRE LES ASPÉRITÉS DU TERRAIN

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.

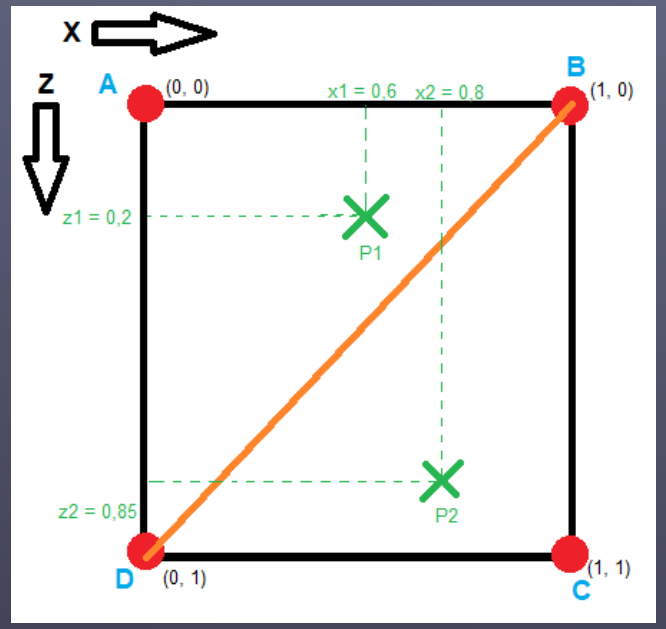
Zoomons sur une des mailles du schéma précédent :



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.

DX barnsten

SUIVRE LES ASPÉRITÉS DU TERRAIN



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.

DX barnsten

SUIVRE LES ASPÉRITÉS DU TERRAIN

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 ☺

```
function Barycentre(p1, p2, p3 : TPoint3D; p4 : TPointF):single;
var
  det, l1, l2, l3, d1, d2, d3,  t1,t2 : single;
begin
  d1 := (p2.z - p3.z);  // Petites optimisations pour ne faire les
  d2 := (p3.x - p2.x);
  d3 := (p1.x - p3.x);
  det := 1 / ((d1 * d3) + (d2 * (p1.z - p3.z))); // Inverse, perme
  t1 := (p4.x - p3.x);
  t2 := (p4.y - p3.z);
  l1  := (( d1 * t1) + (d2 * t2 )) * det;
  l2  := ((p3.z - p1.z) * (t1 + (d3 * t2 ))) * det;
  l3  := 1 - l1 - l2;
  result := l1 * p1.y + l2 * p2.y + l3 * p3.y;
end;
```

SUIVRE LES ASPÉRITÉS DU TERRAIN

Dans le projet FMX Race, la fonction **CalculerHauteur** implémente ce que nous venons de voir et permet de renvoyer la hauteur d'un point à partir :

- de sa position;
- du **TMesh** représentant le relief;
- la taille du **TMesh;**
- la mise à l'échelle par rapport à la hauteur du **TMesh**.

```
function CalculerHauteur(mesh : TMesh; P: TPoint3D; miseAEchelle : single; moitieCarte, sizeMap : integer) : single;
var
   grilleX, grilleZ, sizeMapPlus1 : integer;
   xCoord, zCoord, hauteurCalculee : single; // coordonnées X et Z dans le "carré"
begin
  sizeMapPlus1 := sizeMap + 1;
  // Détermination des indices permettant d'accéder a sommet en fonction de la position du joueur
  grilleX := Math.Floor(P.X+moitieCarte);
  grilleZ := Math.Floor(P.Z+moitieCarte);

  // Si on est en dehors du mesh, on force (arbitrairement) la hauteur à la hauteur de la mer
  if (grilleX >= SizeMap) or (grilleZ >= SizeMap) or (grilleX < 0) or (grilleZ < 0) then
  begin
    result := 0;
  end
  else
  begin
    xCoord := Frac(P.X); // position X dans la maille courante
    zCoord := Frac(P.Z); // position y dans la maille courante

    // On calcule la hauteur en fonction des 3 sommets du triangle dans lequel se trouve le joueur
    // On détermine dans quel triangle on est
    if xCoord <= (1 - zCoord) then
    begin
      hauteurCalculee := Barycentre(TPoint3D.Create(0,-mesh.data.VertexBuffer.Vertices[grilleX + (grilleZ * SizeMapPlus1)].Z,0),
                                    TPoint3D.Create(1,-mesh.data.VertexBuffer.Vertices[grilleX +1+ (grilleZ * SizeMapPlus1)].Z,0),
                                    TPoint3D.Create(0,-mesh.data.VertexBuffer.Vertices[grilleX + ((grilleZ +1)* SizeMapPlus1)].Z,1),
                                    TPointF.Create(xCoord, zCoord));
    end
    else
    begin
      hauteurCalculee := Barycentre(TPoint3D.Create(1,-mesh.data.VertexBuffer.Vertices[grilleX +1+ (grilleZ * SizeMapPlus1)].Z,0),
                                    TPoint3D.Create(1,-mesh.data.VertexBuffer.Vertices[grilleX +1+ ((grilleZ +1) * SizeMapPlus1)].Z,1),
                                    TPoint3D.Create(0,-mesh.data.VertexBuffer.Vertices[grilleX + ((grilleZ +1)* SizeMapPlus1)].Z,1),
                                    TPointF.Create(xCoord, zCoord));
    end;

    hauteurCalculee := hauteurCalculee * miseAEchelle + mesh.Depth*0.5;  // Hauteur calculée et mise à l'échelle
    result := hauteurCalculee;
  end;
end;
```

DX barnsten

SUIVRE LES ASPÉRITÉS DU TERRAIN

La position du joueur suit désormais le relief du terrain. Nous allons donner une meilleure impression du relief en inclinant la voiture en conséquence.

Pour ce faire, je me suis contenté de connaitre la hauteur aux positions de trois des 4 roues de la voiture.
La différence de hauteur entre la roue gauche et la roue droite permettra de déterminer l'angle de la rotation à appliquer sur l'axe Z.
De même, la différence de hauteur entre une roue avant et une roue arrière permettra de déterminer l'angle de rotation à appliquer sur l'axe X.

```
procedure TfPrincipale.InclinaisonVoiture(objet: TModel3D);
var
  hauteurRoueAvantGauche, hauteurRoueAvantDroite, hauteurRoueArriereDroite : single;
begin
  hauteurRoueAvantGauche := CalculerHauteur(mSol, TPoint3D(TModel3D(objet.Children[2].Children[1]).AbsolutePosition), miseAEchelle, moitieCarte, sizemap);
  hauteurRoueAvantDroite := CalculerHauteur(mSol, TPoint3D(TModel3D(objet.Children[2].Children[0]).AbsolutePosition), miseAEchelle, moitieCarte, sizemap);
  hauteurRoueArriereDroite := CalculerHauteur(mSol, TPoint3D(TModel3D(objet.Children[1].Children[0]).AbsolutePosition), miseAEchelle, moitieCarte, sizemap);
  objet.RotationAngle.X := (hauteurRoueAvantGauche - hauteurRoueArriereDroite)*sizemap*scaleVoiture;
  objet.RotationAngle.Z := 180-(hauteurRoueAvantGauche - hauteurRoueAvantDroite)*sizemap*scaleVoiture;
end;
```

DÉTECTION DES COLLISIONS AVEC LES OBSTACLES

Nous allons compléter le décor avec des obstacles de plusieurs sortes :

- des bâtiments;
- des panneaux;
- des piliers;
- des murs.

Ces obstacles sont des objets 3D que nous plaçons comme nous le voulons dans notre décor. Ils seront toutefois tous enfants du **mSol**.
Les obstacles doivent empêcher la voiture de passer. Dans FMX Race, nous n'allons pas gérer de physique particulière : un simple contact avec un obstacle arrêtera brutalement la voiture.

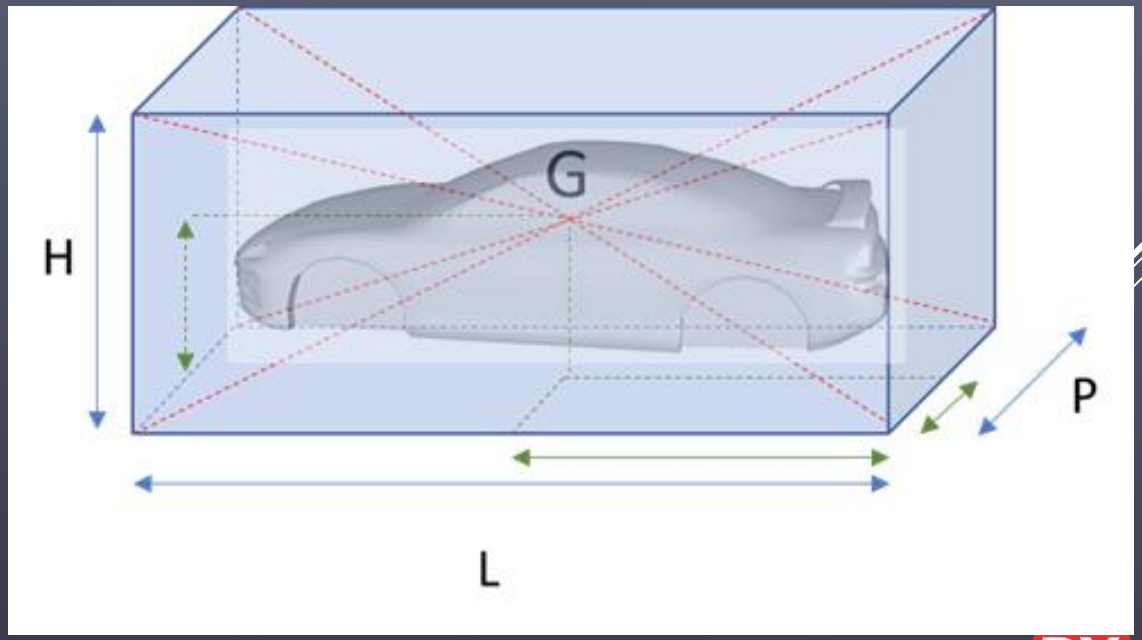
Il va donc falloir être en capacité de reconnaitre les objets à considérer comme obstacle. J'ai choisi d'utiliser la propriété **Tag** des objets : si l'objet a sa propriété **Tag** à 1, alors il sera considéré comme un obstacle.

DÉTECTION DES COLLISIONS AVEC LES OBSTACLES

Pour détecter les collisions, nous allons utiliser la technique des « bounding box » : nous allons détecter une collision entre la « boite » imaginaire englobant la voiture et la « boite » englobant l'obstacle.

Cette technique n'est pas précise, mais elle est simple à mettre en œuvre et suffisante pour notre exemple.

Le schéma ci-contre montre en bleu
la boite imaginaire qui contient l'objet.

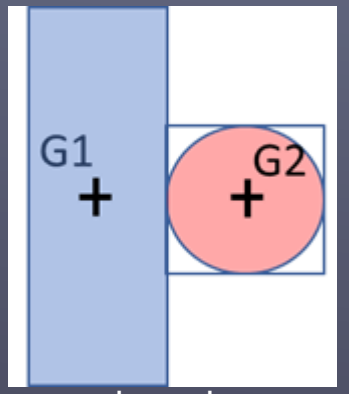


DÉTECTION DES COLLISIONS AVEC LES OBSTACLES

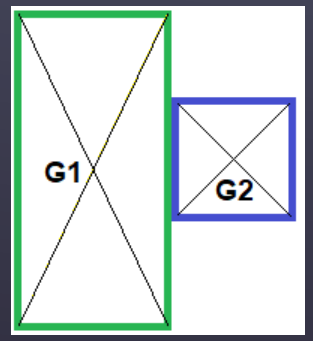
La détection des collisions va simplement consister à calculer la distance entre le centre de la voiture et le centre de l'obstacle.

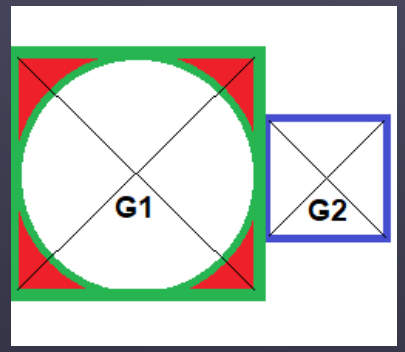
Passons temporairement en 2D.

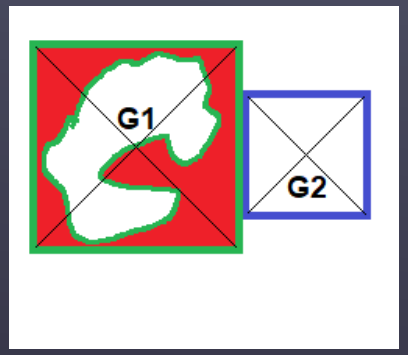
Sur ce schéma, nous constatons qu'il y a collision entre l'objet bleu et l'objet rouge lorsque la distance entre le centre G1 et le centre G2 est inférieure ou égale à la demi largeur de l'objet bleu et la demi largeur de l'objet rouge.



La précision de la détection n'est pas optimum. Les schémas ci-dessous montrent en rouge la perte de précision en fonction de la forme des objets :







DX barnsten

DÉTECTION DES COLLISIONS AVEC LES OBSTACLES

La fonction **DetectionCollisionObstacle** va effectuer ce calcul.

Cette fonction prend en paramètre un **TMesh** et un **TControl3D**. Elle retourne un booléen indiquant s'il y a collision ou non.

On parcourt tous les enfants du **TMesh** à la recherche de ceux ayant leur propriété **Tag** à 1.
On calcul alors la distance entre le **TControl3D** passé en paramètre et le composant enfant du **TMesh**.

```
function DetectionCollisionObstacle(mesh : TMesh; objet : TControl3D):boolean;
var
  unObjet3D:TControl3D; // l'objet en cours de rendu
  DistanceEntreObjets,distanceMinimum: TPoint3D;
  i : integer;
  resultat : boolean;
begin
  resultat := false;
  // Test collision avec enfants directs de mSol
  for I := 0 to mesh.ChildrenCount-1 do
  begin
    if mesh.Children[i].Tag = 1 then
    begin
      // On travail sur l'objet qui est en train d'être calculé
      unObjet3D := TControl3D(mesh.Children[i]);
      DistanceEntreObjets := unObjet3D.AbsoluteToLocal3D(TPoint3D(objet.AbsolutePosition)); // Distance entre l'objet 3d
      distanceMinimum := (SizeOf3D(unObjet3D) + SizeOf3D(objet)) * 0.5; // distanceMinimum : on divise par 2 car le cent

      // Test si la valeur absolue de position est inférieure à la distanceMinimum calculée sur chacune des composantes
      if ((Abs(DistanceEntreObjets.X) < distanceMinimum.X) and (Abs(DistanceEntreObjets.Y) < distanceMinimum.Y) and
          (Abs(DistanceEntreObjets.Z) < distanceMinimum.Z)) then
      begin
        resultat := true;
        break;
      end;
    end;
  end;

  result := resultat;
end;
```

DÉTECTION DES COLLISIONS AVEC LES OBSTACLES

La fonction **DetectionCollisionObstacle** utilise la fonction **SizeOf3D** qui renvoie la taille de l'objet passé en paramètre.

```
function SizeOf3D(const unObjet3D: TControl3D): TPoint3D;
begin
  Result :=NullPoint3D;
  if unObjet3D <> nil then
    result := Point3D(unObjet3D.Width, unObjet3D.Height, unObjet3D.Depth);
end;
```

Comme indiqué, nous nous contenterons de cette technique de détection pour le projet FMX Race.

DX barnsten

BONUS

Nous venons de voir les techniques 3D principales utilisées dans FMX Race. Ce qui suit concerne plus le déroulement du jeu.

Tout d'abord, j'utilise une boucle principale représentée par un **TFIaotAnimation** nommé **aniPrincipale**. Cette animation va rendre la scène régulièrement et, en fonction de la valeur de la variable **scene**, afficher telle ou telle scène du jeu.

Le jeu est composé de 6 scènes :

- l'introduction;
- la sélection de voiture;
- la saisie du nom du joueur;
- le jeu proprement dit;
- les options;
- les crédits.

Plusieurs caméras sont disponibles pour suivre l'action selon différents points de vue.

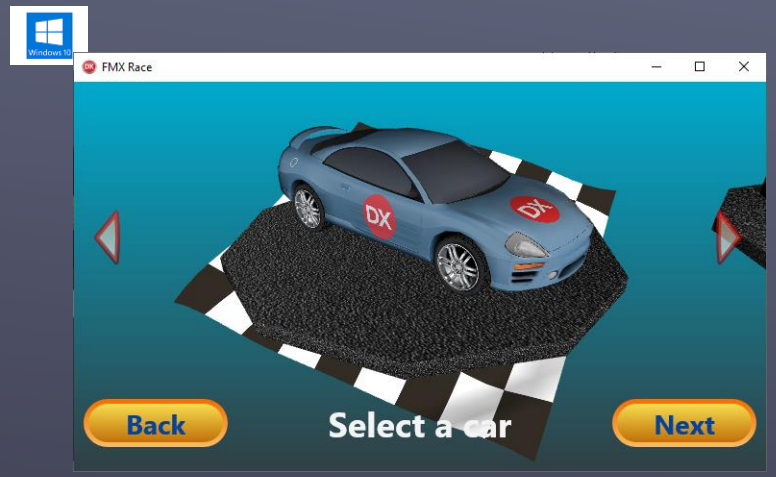
BONUS

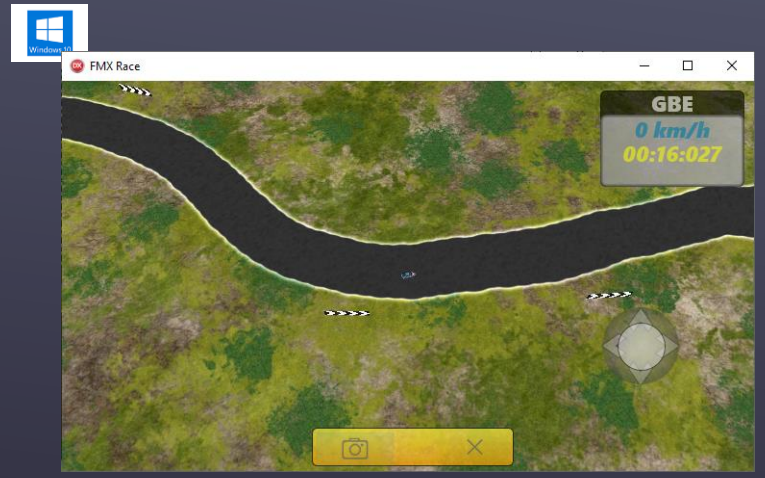
A part la scène de jeu, les autres scènes affichent un drapeau d'arrivée animé en fond d'écran.

Il s'agit d'un **TPlane** subdivisé et pour chaque sommet on modifie la hauteur du sommet à l'aide d'une fonction sinusoïdale.

BONUS

Pour terminer, voici quelques captures d'écran de FMX Race sur plusieurs OS :





Avez-vous des questions ?

DX barnsten

Merci ☺

Grégory Bersegeay

Site web : http://www.gbesoft.fr
Mail : gregory.bersegeay@gbesoft.fr
GitHub : https://github.com/gbegreg
Les sources du projet FMX Race sont disponibles sous https://github.com/gbegreg/FMXRace
Mon pseudo sur Developpez.com : gbegreg