

Delphi tournée européenne 2017

Étape à Paris le 17 mai

Exemple de jeu en 3D multiplate-forme
avec Delphi Tokyo & Firemonkey

Grégory Bersegeay



- 1) Présentation
- 2) Premiers pas en 3D avec Firemonkey
 - pré-requis 3D pour FMX
 - tour d'horizon des composants gérant la 3D
- 3) Édition de la scène 3D graphiquement sous Delphi
- 4) Les animations
- 5) L'éclairage
- 6) Les caméras
- 7) Un peu de géométrie :
 - les collisions
 - déplacement de la balle de manière réaliste
- 8) Mécanismes du jeu
- 9) Utilisation d'un capteur : le capteur de mouvement
- 10) Déploiement sur plusieurs plate-formes

Présentation (1/2)

Salarié dans une grande mutuelle d'assurance française depuis 1998, j'ai eu mon premier contact avec Delphi (version 1) à ce moment là. Un an après, il y a eu la migration à Delphi 5. Aujourd'hui encore, nous utilisons Delphi 5.

En parallèle, je suis auto entrepreneur depuis 2009 et je réalise des développements avec Delphi (principalement dans le domaine de l'indexation de documents, « l'OCRisation » de documents et l'analyse automatique de factures). Je travaille uniquement avec la société **Archivages Services**.

Je suis également l'auteur de l'éditeur de fichier texte gratuit nommé **GBEPad** et il y a quelques mois, j'ai réalisé un tutoriel sur la 3D avec Firemonkey sur le site **Developpez.com**.

J'ai une assez bonne connaissance de la VCL et je me forme de manière autodidacte à Firemonkey.

Site internet : <http://www.gbsoft.fr>

GitHub : <https://github.com/gbegreg/>

Pseudo **gbegreg** sur <https://www.developpez.com>

Présentation (2/2)

Cette présentation va nous conduire à la réalisation d'un petit jeu en 3D multi plate-formes grâce à Delphi et Firemonkey (FMX pour la suite du document).

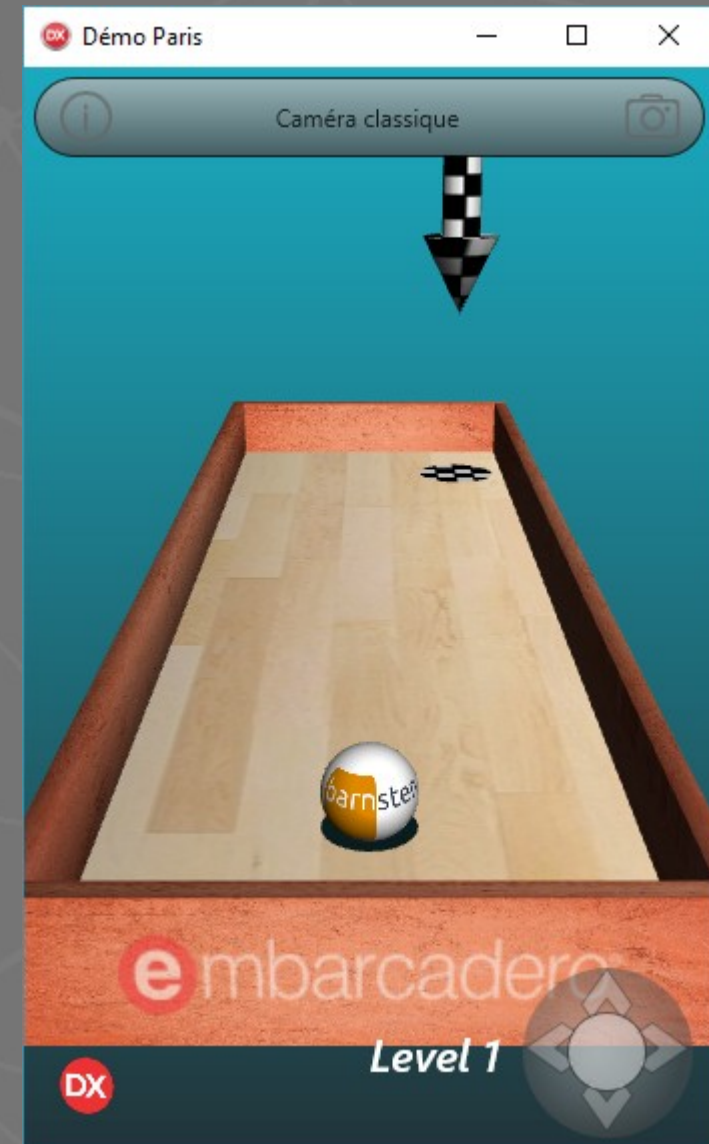
Le principe du jeu est très simple : il faut incliner un plateau afin de diriger une balle sur la zone d'arrivée matérialisée par un cercle.

Pour incliner le plateau, le joueur pourra :

- Utiliser les flèches du clavier sur PC et Mac
- Utiliser le joystick virtuel (toutes plates-formes)
- Utiliser le gyroscope (pour les périphériques équipés)

Ce projet va nous permettre d'aborder les principales fonctionnalités fournies par FMX dans le domaine de la 3D. Nous verrons également l'utilisation d'un capteur ainsi que le déploiement de l'application sur Windows, Mac OS et Android. Pourquoi pas d'IOS ? Car je n'ai pas de périphérique adéquat...

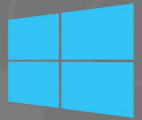
Les sources du projets sont disponibles sur GitHub à l'adresse : <https://github.com/gbegreg/demoParis/>



Capture d'écran du projet final

Premiers pas en 3D avec FMX – Pré requis

Nous allons utiliser les fonctionnalités 3D de FMX. Quelques pré-requis matériels et logiciels sont à respecter :




Windows 10

Sous Windows, FMX s'appuie sur DirectX 11 et le Shader model 5. Par défaut, FMX active l'accélération matérielle. Il faut donc disposer d'une carte graphique compatible sinon des problèmes d'affichage seront probables (par exemple les effets de lumière seront impossibles). Sous l'IDE, la scène 3D s'affiche correctement mais lorsqu'on exécute l'application, la scène 3D est en rouge...

En fait, sous l'IDE, l'accélération matérielle est désactivée. Il est possible de désactiver l'accélération matérielle dans le code en ajoutant la ligne suivante dans le code source du projet (et **avant** l'instruction « application.initialize; ») :

```
fmx.types.GlobalUseDXSoftware := True;
```

L'émulation logicielle effectuera alors le rendu graphique : ça sera moins fluide mais il n'y aura pas de problème.



Sous Mac OS, FMX utilise Quartz (pour la partie 2D) qui lui même s'appuie sur OpenGL pour la partie 3D. Je n'ai pas rencontré de problème particulier de rendu sur Mac.



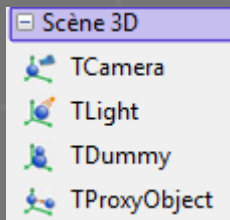
Sous Android, FMX utilise OpenGL. Là non plus, je n'ai pas rencontré de problème particulier de rendu.

Tour d'horizon des composants FMX 3D

Firemonkey fournit un certain nombre de composants pour réaliser une application en 3D.

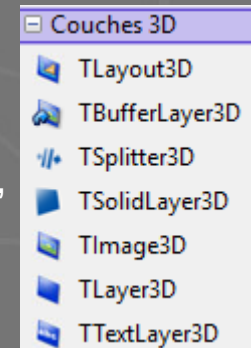
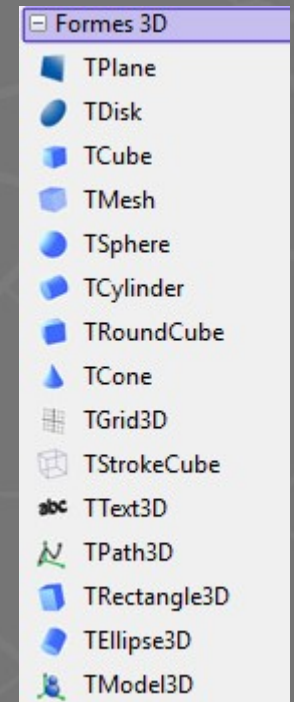
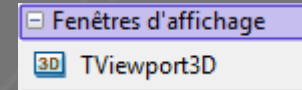
Tout d'abord, le **TViewport3D** que l'on trouve dans la rubrique **Fenêtres d'affichage**, est le composant de base. Il est indispensable car c'est le conteneur de toute la scène 3D.

Ensuite, nous avons dans la rubrique **Formes 3D**, les différents objets 3D disponibles : objets géométriques (plan, cube, sphère, cône, cylindre...) mais également des objets plus complexes permettant par exemple de charger des objets 3D comme le TModel3D (supportant les formats .obj, .dae et .ase).



Autre rubrique essentielle, la rubrique **Scène 3D** : c'est ici que l'on retrouve les composants gérant les caméras (le TViewport3D dispose d'une caméra par défaut), les lumières et d'autres composants que l'on verra plus loin.

Enfin, la rubrique **Couches 3D** fournit le nécessaire pour, par exemple, afficher des composants 2D dans la scène 3D.



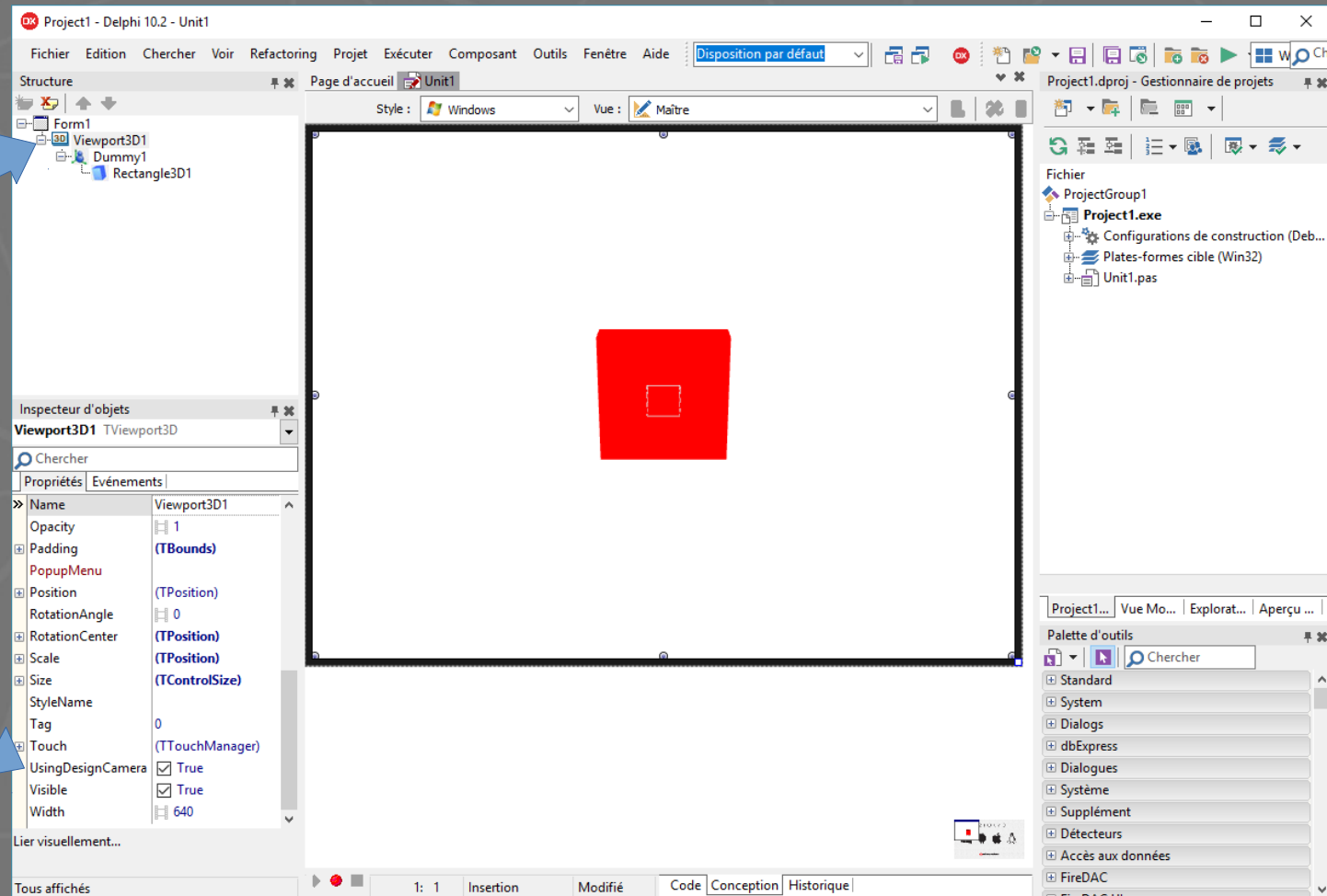
Édition de la scène 3D depuis Delphi (1/4)

C'est parti ! Démarrons un nouveau projet de type **Application multi-périphérique**.

Plaçons un **TViewport3D**, puis un **TDummy** et enfin un **TRectangle3D**. Une des spécificités de Firemonkey est que n'importe quel composant peut être parent/enfant d'un autre.

Ainsi, la vue **Structure** permet de modifier la hiérarchie des composants.

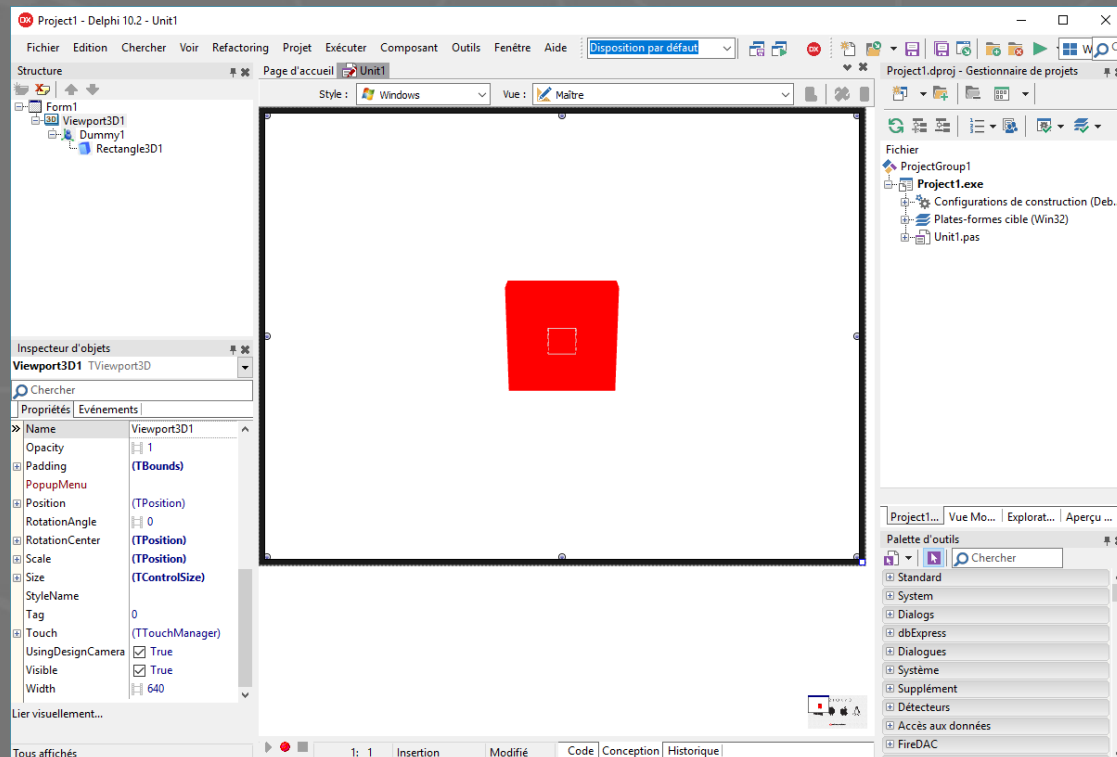
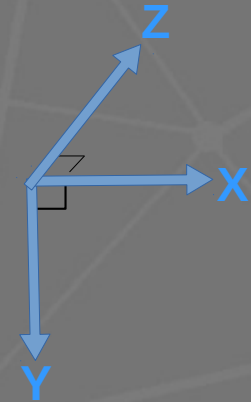
A noter : on utilise la caméra par défaut du Viewport.



Édition de la scène 3D depuis Delphi (2/4)

Décrivons la petite scène 3D que nous venons de créer.

Tout d'abord, le repère orthonormé est orienté par défaut de la manière suivante : Il faut bien avoir ce repère en tête... En effet, tous les objets 3D disposent d'une propriété **point** de type **TPoint3D** qui lui même est composé de 3 propriétés de type **single** nommées X, Y et Z.



De plus, la caméra par défaut du **TViewport3D** nous montre la scène avec une rotation de 20° autour de l'axe X vers nous.

Le **TDummy** n'est visible sous forme de cube en pointillé que lors de la conception. A l'exécution, le TDummy n'est pas visible. Il est cependant très utile car il nous permettra d'effectuer des transformations (translations, rotations, étirements...) sur tous les objets 3D qui lui seront rattachés.

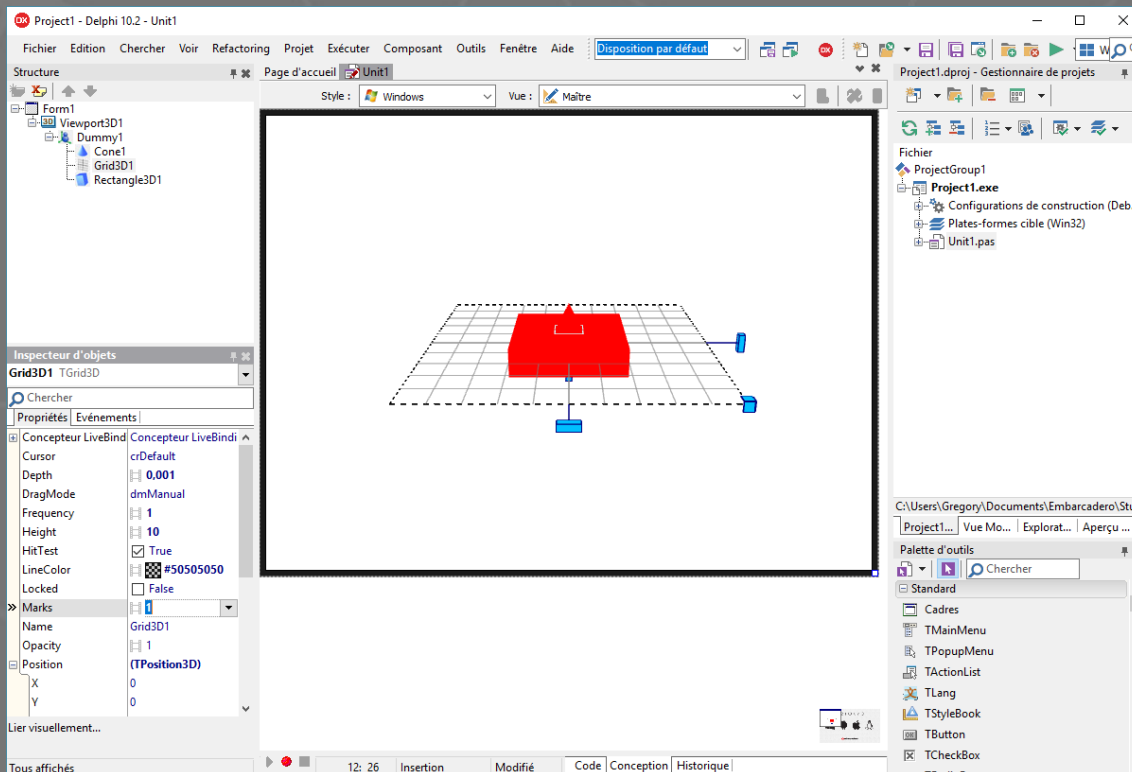
Le **TRectangle3D** apparaît en rouge : il s'agit de la couleur par défaut.

Édition de la scène 3D depuis Delphi (3/4)

Plusieurs propriétés communes à tous les objets 3D sont disponibles pour tailler (largeur, hauteur, épaisseur), positionner, étirer et faire pivoter l'objet comme on le souhaite.

Important : la position de l'objet 3D est en fait la position du point qui est au centre de l'objet.

Dans notre exemple, nous allons ajouter un **TCone** et un **TGrid3D** que l'on va rendre enfant du **TDummy**. Ensuite, nous allons faire pivoter le rectangle et la grille de 90° sur l'axe X. Enfin, on va placer le cône à la position X=0, Y=-1 et Z=0.



Le concepteur Delphi permet également de positionner les objets 3D de manière simple c'est à dire à la souris et avec les petites poignées bleues.

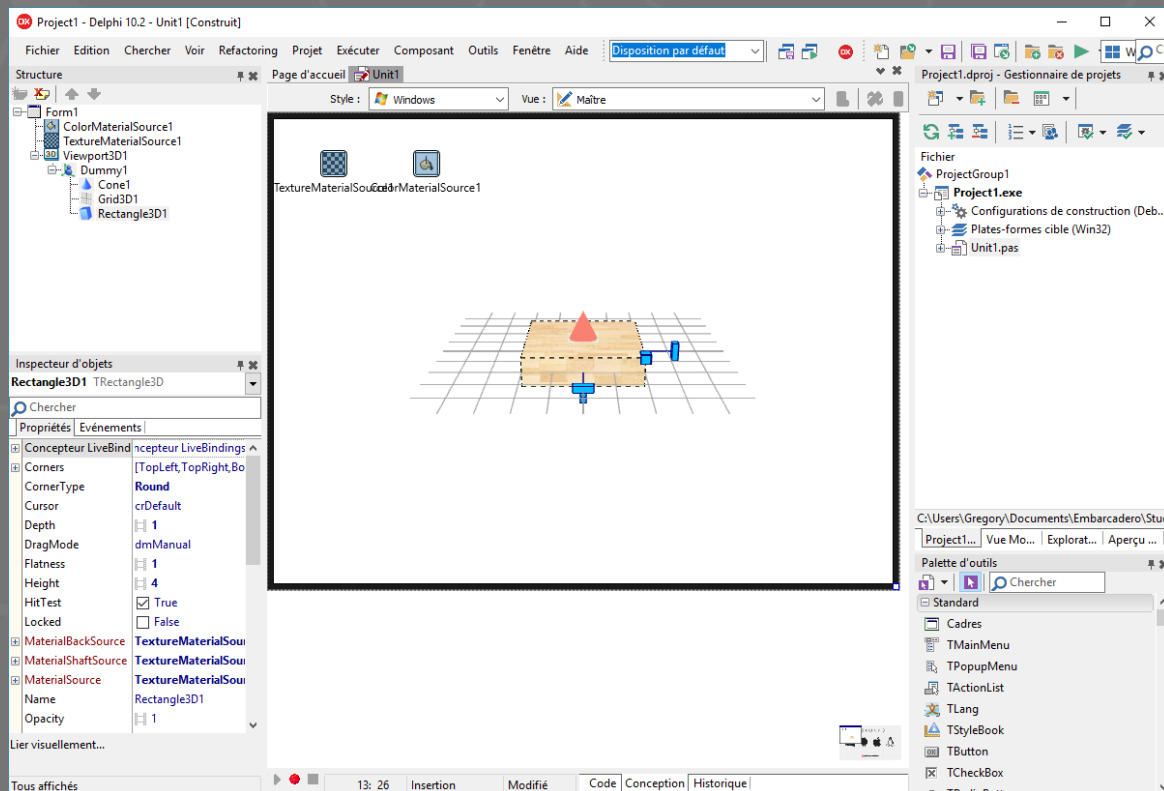
Pour modifier la couleur d'un objet 3D, il faut utiliser un **material**. Il y a 3 types de **material** :

- **TColorMaterialSource** : il permet de choisir une couleur unie.
- **TTextureMaterialSource** : il permet d'appliquer une texture.
- **TLightMaterialSource** : c'est le material le plus complexe. Il permet d'appliquer une texture ou une couleur et de définir la manière dont il va réagir à la lumière.

Édition de la scène 3D depuis Delphi (4/4)

Nous allons utiliser un **TTextureMaterialSource** et un **TColorMaterialSource** pour coloriser un peu la scène.

On choisit une couleur pour le **TColorMaterialSource** et une image pour le **TTextureMaterialSource**. Ensuite, via l'inspecteur d'objet, on associe le **TCone** avec le **TColorMaterialSource** via sa propriété **Material**.



Nous faisons de même pour associer le **TTextureMaterialSource** au **TRectangle3D**. Cependant, l'objet rectangle dispose de 3 propriétés material :

- **MaterialBackSource** : il s'agit du material qui sera utilisé pour coloriser la face arrière du rectangle
- **MaterialShaftSource** : le material qui servira à coloriser la tranche du rectangle.
- **MaterialSource** : le material qui servira pour la face avant du rectangle.

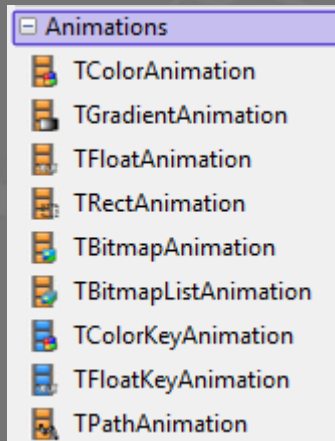
Nous associons le même material aux 3 propriétés.

Les animations (1/4)

Nous venons de créer une petite scène 3D sans écrire une seule ligne de code. Nous allons maintenant l'animer. Pour ce faire, plusieurs possibilités sont envisageables :

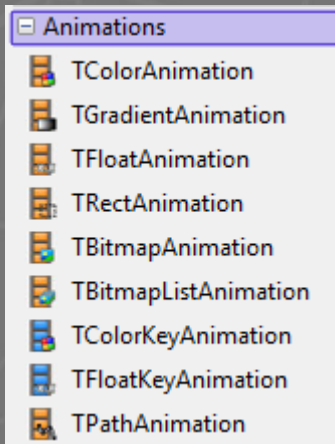
- utilisation d'un Timer et coder dans l'événement OnTimer les modifications à apporter à la scène.
- utilisation des animations fournies par Firemonkey. C'est cette solution que nous allons voir en trois temps :
 - via les composants et utilisation des propriétés des composants (sans une ligne de code)
 - via les composants et du code pour des animations plus complexes
 - création dynamique d'animation

Les composants gérant les animations sont dans la rubrique **Animations**. Toutes les animations se déroulent sur une durée et peuvent se déclencher après un certain délais, tourner en boucle, s'inverser...

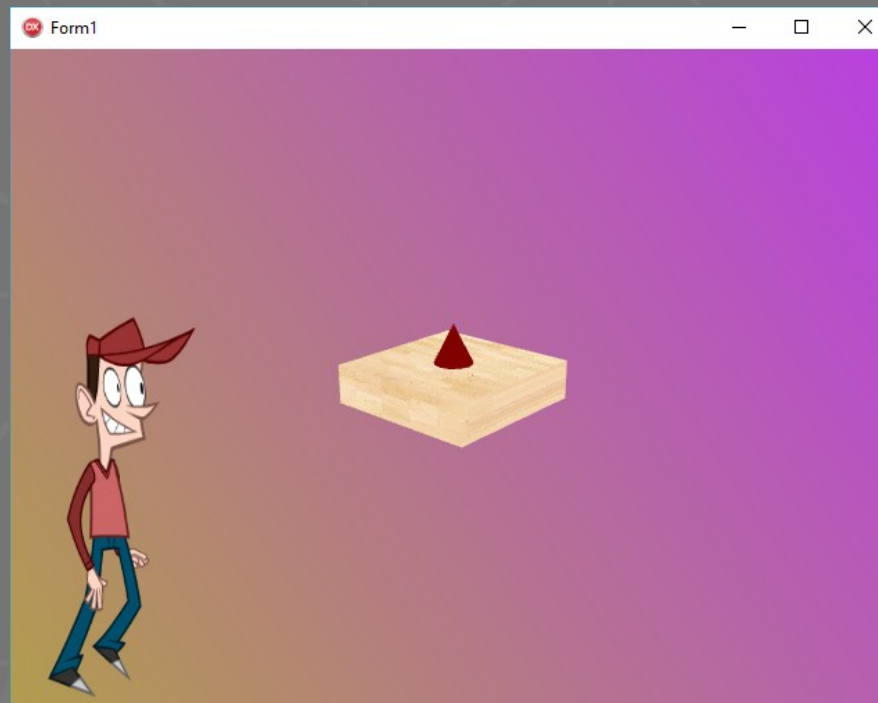


- **TColorAnimation** : permet de passer d'une couleur unie à une autre la propriété Color d'un objet.
- **TGradientAnimation** : permet de modifier les couleurs d'une propriété gradient (dégradé) d'un objet.
- **TFloatAnimation** : permet de modifier les valeurs des propriétés de type flottant d'un objet (exemple : les positions, les angles de rotation...).
- **TRectAnimation** : permet de modifier les valeurs des propriétés de type TBounds d'un objet (exemple : margins et padding).
- **TBitmapAnimation** : permet de modifier la propriété Bitmap d'un objet (exemple d'utilisation : faire un diaporama).
- **TBitmapListAnimation** : lorsqu'un Bitmap contient en fait un ensemble de « vignettes » constituant une animation, le TbitmapListAnimation permet de n'afficher qu'une vignette à la fois (exemple d'utilisation : faire des sprites).

Les animations (2/4)



- **TColorKeyAnimation** : permet de passer d'une couleur unie à une autre la propriété Color d'un objet. Comme le TColorAnimation mais là on peut spécifier une liste de couleurs.
- **TFloatKeyAnimation** : comme le TFloatAnimation mais en indiquant une suite de valeurs. Cela permet de créer des animations plus complexes qui ne font pas qu'incrémenter/décrémenter une valeur de propriété.
- **TPathAnimation** : permet de spécifier un chemin prédéfini. L'objet sera déplacé en suivant les étapes définies dans la propriété Path.



Petit exemple avec plusieurs animations

Les animations (3/4)

Nous venons de voir les composants gérant les animations et leurs propriétés via l'inspecteur d'objet. Il est possible d'aller plus loin dans les animations mais, il va falloir coder un peu...

Les composants d'animation disposent des événements **OnFinish** et **OnProcess**.

OnFinish se déclenche lorsque l'animation se termine et permet donc de réaliser une action à la fin de l'animation. Une animation se termine lorsque la durée spécifiée est atteinte (propriété **Duration** de l'animation : attention si la propriété **Loop** est à true, à la fin de la durée spécifiée, l'animation reprend au départ et on ne passe pas dans le OnFinish) ou lorsque, dans le code, il y a l'instruction « *monAnimation.stop ;* »

OnProcess se déclenche dès que l'animation est activée : propriété **Enabled** de l'animation à true ou via le code « *monAnimation.start ;* ». C'est dans cet événement que l'on peut coder plus finement l'animation.

```
procedure TForm1.FloatAnimation1Process(Sender: TObject);
39 begin
40     Dummy1.RotationAngle.Z := Dummy1.RotationAngle.Z + 1;
    Dummy1.RotationAngle.X := Dummy1.RotationAngle.X + 2;

    Dummy1.Position.X := Dummy1.Position.X + 0.02;
    Dummy1.Opacity := Dummy1.Opacity - 0.002;
end;
```

Dans cet exemple, le FloatAnimation1 est paramétré via l'inspecteur d'objet pour effectuer une rotation autour de l'axe Y, dans le **OnProgress**, on effectue en plus une rotation d'un degré sur l'axe Z, de deux degrés sur l'axe X tout en déplaçant l'objet sur l'axe X et en le rendant transparent (on diminue son opacité).

Les animations (4/4)

Dernier point de ce survol des animations, il est possible d'en créer dynamiquement sans passer par les composants.

Cette technique peut être utilisée pour effectuer des effets éphémères. On peut imaginer par exemple un bouton qui s'anime lorsque l'utilisateur clique dessus : par exemple faire un petit effet de zoom lors du clic sur un bouton, changer la couleur lors du survol d'un composant par le pointeur de souris etc.

Cela se fait simplement en une ligne de code.

Exemple :

```
TAnimator.AnimateFloat(Image3D1, 'Position.X', 4, 5);
```

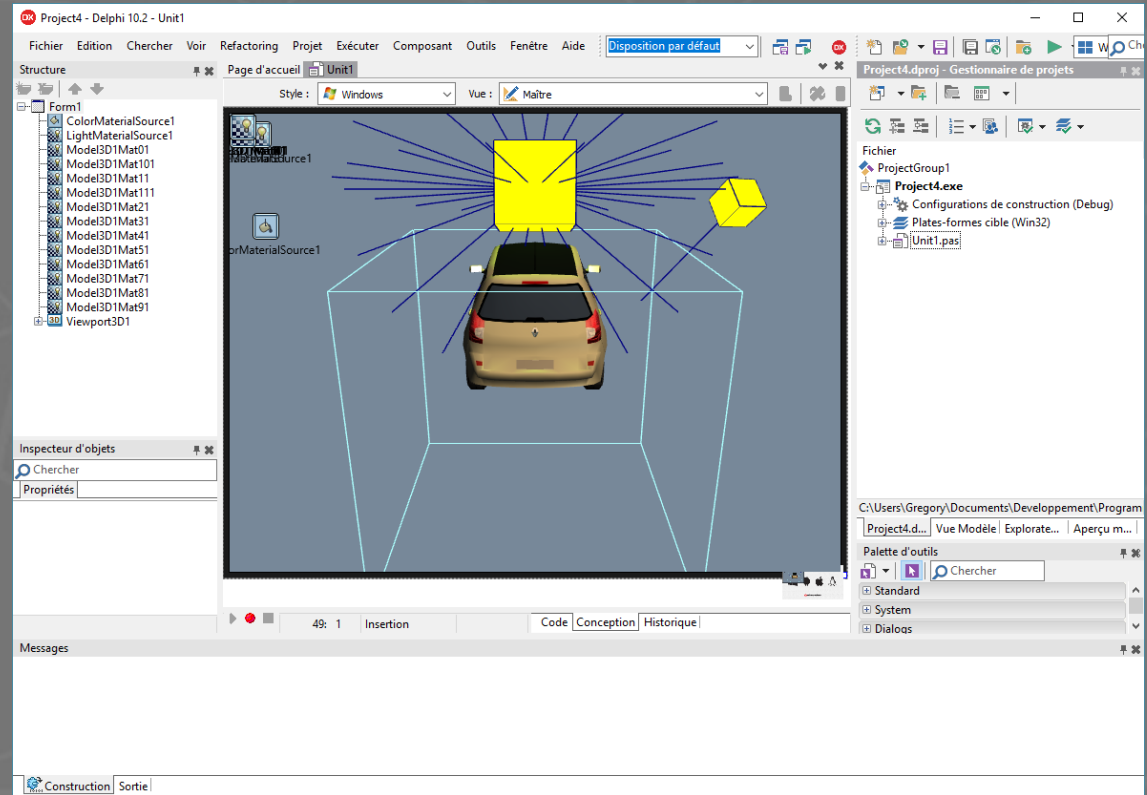
Dans ce cas, l'animation va déplacer l'objet Image3D1 sur l'axe X de sa position actuelle jusqu'à X = 4 sur une durée de 5 secondes. L'animation est fluide et sans scintillement.

L'éclairage (1/2)

Pour ajouter toujours plus de réalisme, Firemonkey permet de mettre en lumière la scène 3D. Pour cela, nous avons à notre disposition le composant **TLight**.

Ce composant représente une source de lumière qui va agir avec les **TLightMaterial**. Il est bien sûr possible de choisir la couleur de la source lumineuse, mais également le type de lumière parmi 3 types :

- **Directional** : tous les rayons sont parallèles et éclairent la scène de manière uniforme en fonction d'une direction donnée.
- **Point** : les rayons partent de la source et se propagent dans toutes les directions.
- **Spot** : les rayons partent de la source, se propagent dans la direction donnée sous forme de cône. La puissance du spot est paramétrable.



Les TLight étant des objets comme les autres, il est possible d'agir sur eux, de leur appliquer des animations...

A noter :

- Si on charge un objet 3D dans un TModel3D, les textures associées à l'objet 3D seront traitées en tant que TLightMaterial et réagiront donc aux lumières.
- Comme indiqué en pré-requis, la gestion de l'éclairage par FMX sous Windows utilise DirectX 11 et le shader model 5. Il se peut que sur d'anciens processeurs graphiques le rendu des couleurs ne soit pas celui escompté. Il est possible de forcer FMX à faire le rendu graphique logiciel (sans s'appuyer sur les capacités du processeur graphique). Le résultat est ainsi correct mais les performances peuvent s'en ressentir...

Enfin, dernier objet important les TCamera sont des objets permettant de définir des points de vue différents d'une même scène 3D.

Elles peuvent être fixes, en mouvement ou suivre un objet précis.

Pour activer une caméra, il faut la sélectionner dans la propriété **Camera** du composant **TViewport3D** et veiller à ce que la propriété **UsingDesignCamera** soit à false.

Plusieurs caméras seront disponibles dans le jeu présenté ensuite.

Un peu de géométrie (1/4)

Nous avons fait un tour d'horizon rapide de ce que fournissait FMX. Ce sont ces briques qui vont nous servir à faire notre jeu.

Cependant, il va falloir faire un peu de géométrie pour :

- détecter les collisions
- gérer de manière réaliste les mouvements de la balle

Rassurez vous, il n'y aura rien de compliqué car c'est un projet qui se veut pédagogique et ludique : j'ai essayé de simplifier la partie mathématique.

Par exemple pour la gestion des collisions, les obstacles sont forcément immobiles et parallèles ou perpendiculaires aux côtés du plateau (\Rightarrow ils ne peuvent pas être « obliques »).

Un peu de géométrie (2/4)

Détection de collisions

Dans notre jeu, la balle va se déplacer sur les axes X et Z. Le plateau étant plat, il n'y aura pas d'évolution sur l'axe Y.

Nous allons donc simplifier la gestion des collisions en considérant uniquement les positions de la balle et des obstacles sur les axes X et Z.

X4 = Position.X de l'obstacle - 1/2 largeur de l'obstacle

X3 = Position.X de l'obstacle + 1/2 largeur de l'obstacle

Z3 = Position.Z de l'obstacle + 1/2 profondeur de l'obstacle

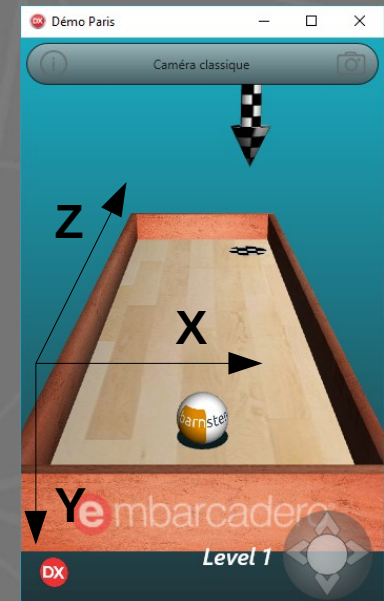
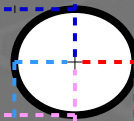
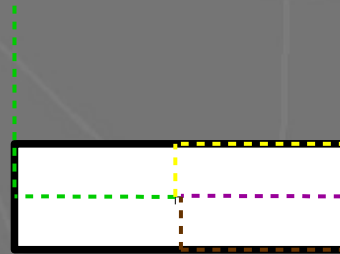
Z4 = Position.Z de l'obstacle - 1/2 profondeur de l'obstacle

Z1 = Position.Z de la balle + 1/2 profondeur de la balle

Z2 = Position.Z de la balle - 1/2 profondeur de la balle

X2 = Position.X de la balle - rayon de la balle

X1 = Position.X de la balle + rayon de la balle



La détection (très) simplifiée consiste donc à contrôler que X1 ou X2 ne se trouvent pas entre X3 et X4 et que dans le même temps Z1 ou Z2 ne se trouvent pas entre Z3 et Z4.

Pour info : si on souhaite réaliser de véritables tests de collisions, il faut regarder du côté des méthodes RayCastIntersect disponibles sur les objets 3D.

Un peu de géométrie (3/4)

Mouvements de la balle

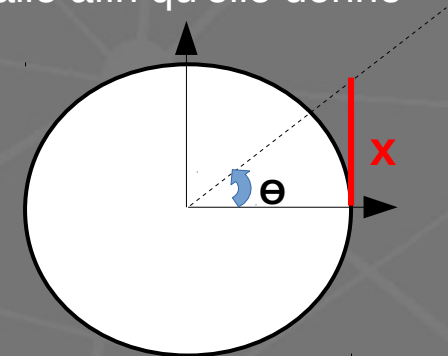
La balle doit se déplacer en fonction de l'inclinaison du plateau. Pour ce faire, nous n'allons pas entrer dans des calculs complexes de trajectoire : nous allons utiliser les animations de FMX.

Lorsque le plateau s'incline vers la droite ou la gauche, la balle se déplacera vers la droite ou la gauche sur l'axe X. De même, lorsque le plateau s'incline vers le joueur ou le fond, la balle se déplacera vers le joueur ou le fond sur l'axe Z.

Dans ces quatre situations, nous utiliserons une **TFloatAnimation** pour déplacer la balle ce qui permettra de donner une trajectoire réaliste à la balle.

Problème supplémentaire, la balle n'est pas de couleur unie : une texture lui est appliquée. Afin que le mouvement de la balle soit réaliste, il faut appliquer des rotations à la balle afin qu'elle donne l'impression de rouler sur le plateau.

Pour déterminer l'angle de rotation nécessaire à donner à un cercle pour parcourir une distance donnée, il faut utiliser la fonction arctan : $\Theta = \arctan(x)$. Cette fonction renvoi l'angle en radian, il faut donc convertir le résultat en degré via la fonction RadToDeg.



Dans le jeu, j'ai voulu décomposer la rotation à donner à la sphère en appliquant des rotations successives : une rotation autour de l'axe X pour les déplacements en profondeur et en Z pour les déplacements droite/gauche. Mais ça ne fonctionne pas : la première rotation fait également pivoter le repère de la sphère.

J'ai adopté une autre solution mais le rendu a des défauts...

A chaque inclinaison du plateau, je calcule l'angle X et l'angle Z. Pour appliquer la rotation résultante de la balle, j'initialise la rotation de la balle à sa valeur initiale et je les applique les angles calculés en une seule fois via la propriété **point** du paramètre **RotationAngle** de la sphère.

Mécanismes du jeu

Nous avons vu les points techniques. Il reste à présenter l'ossature du jeu.

Le jeu disposera d'un écran d'accueil, d'une boîte de dialogue de paramétrage et de la vue jeu elle-même.

Une boucle principale sous forme d'une `TFloatAnimation` sera le cœur du programme : elle permettra d'enchaîner les écrans et les niveaux, de gérer les mouvements, les collisions...

En plus d'utiliser les flèches directionnelles, un petit joystick virtuel est disponible pour incliner le plateau.

Le jeu est fourni avec 3 niveaux différents. Ces derniers sont constitués en dur via le code.

Plusieurs caméras sont disponibles pour visualiser l'action sous différents angles.

Enfin, une boîte de dialogue permet d'afficher certains paramètres et de faire persister les choix de l'utilisateur d'une partie à l'autre.

Utilisation d'un capteur : le capteur de mouvement

Petit bonus, le jeu peut utiliser le capteur de mouvement pour incliner le tableau. Évidemment, le périphérique doit être équipé d'un tel capteur.

D'un point de vue développement, Delphi simplifie là encore les choses puisqu'il suffit de disposer sur la form un **TMotionSensor**.

Une fois le composant activé, il suffit de lire les propriétés **Sensor.AccelerationX** et **Sensor.AccelerationZ** pour incliner le plateau en fonction.

J'ai également utilisé un **TTrackBar** afin de pouvoir régler la sensibilité du capteur.

Déploiement sur plusieurs plate-formes

Delphi permet de déployer une application sur Windows, Mac OS, Android et IOS.

Nous allons donc déployer le jeu sur Windows, Mac OS et Android (je n'ai pas de périphérique IOS).

Il y a quelques petites différences d'une plate-forme à l'autre. Dans ce projet, j'ai noté 3 différences pour ce projet :

- Le **TText3D** d'introduction est plus gros sur Mac OS que sous les autres plate-formes
- Les images des **TSpeedButton** n'ont pas le même rendu sur Android
- L'utilisation du capteur de mouvement étant actif, il est conseillé de forcer l'orientation de l'application pour Android. Sans cela, il faut gérer la orientation dans le code et du coup adapter les propriétés du capteur de mouvement à prendre en compte.

Pour gérer ces différences, j'ai utilisé dans le code les directives **`{$IF DEFINED(MACOS)}`** et **`{$IF DEFINED(Android)}`**.

Je vous remercie pour votre attention et n'hésitez pas si vous avez des

questions