Hands-On Lab

Aller plus loin avec XNA

Version: 1.0.0

Dernière mise à jour : 1/10/2012

Sommaire

Vue d'ensemble

Cet atelier va vous permettre d'expérimenter le développement avancé d'une application XNA, en vous expliquant les principes de base de la 3D et de leur utilisation dans le cas d'une application Windows Phone, en allant du dessin de formes sommet par sommet jusqu'au chargement et l'affichage d'un modèle.

Objectifs

A la fin de cet atelier, vous aurez appris :

- Les bases fondamentales de la 3D et la terminologie associée
- Comment utiliser XNA pour afficher des formes 3D
- Comment appliquer une texture sur un objet
- Comment ajouter de la lumière à une sècne
- Comment charger un modèle 3D exporté depuis un logiciel de modélisation et comment l'animer

Pré requis

Afin de mener à bien cet atelier, vous devez installer les éléments suivants :

- Microsoft Visual Studio 2010 Express pour Windows Phone 7

Exercices

Cet atelier est divisé en plusieurs exercices. Ceux-ci sont listés ci-dessous :

1. Dessiner une forme avec XNA
2. Dessin d'une forme plus complexe avec XNA
3. Appliquer une texture à une forme 3D
4. Animer on objet
5. Ajouter de la lumière à une scène
6. Charger et afficher un modèle 3D

Durée estimée pour compléter cet atelier: **90 minutes**.

Atelier : Aller plus loin avec XNA

Microsoft® XNA™ Game Studio 4.0 est un produit vous permettant de développer très simplement vos propres jeux vidéo. Tout développeur, qu'il soit étudiant, passionné ou professionnel peut utiliser cet outil pour développer des jeux et partager ses créations.

Le XNA Game Studio 4.0 est un outil fournit par Microsoft et pensé pour être utilisé avec Visual Studio 2010 Express pour Windows Phone, permettant ainsi aux développeurs d'allier la puissance et la simplicité du langage C# pour développer des jeux vidéo.

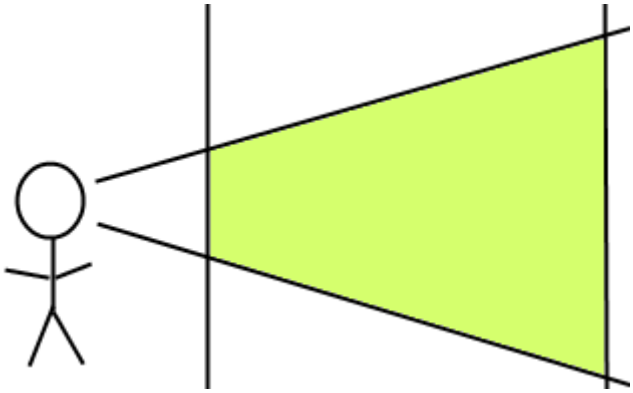
XNA Game Studio 4.0 embarque le Framework XNA en tant que tel ainsi que le XNA Framework Content Pipeline qui permet d'importer et d'utiliser très simplement des modèles 3D, texture, sons ou autres médias au sein de vos jeux. Il fournit également une API (Application Programming Interface) orientée pour le développement de jeux vidéo, permettant ainsi de simplifier et unifier les développements pour ses différentes plateformes cibles : Xbox 360®, Windows® et depuis peu **Windows Phone 7 Series®**.

Tout au long des exercices de cet atelier, vous apprendrez les principes du développement d'applications 3D avec XNA pour la création d'une application de réalité augmentée utilisant également les fonctions de géolocalisation du téléphone.

Principes de base de la 3D

Pour bien débuter en 3D, il est important de comprendre la notion de projection.

Dans le schéma ci-dessous, la zone en verte représente la zone dans laquelle les objets peuvent être dessinés à l'écran. C'est la matrice de projection. Elle est délimitée par un « near plan », représenté par la ligne verticale à gauche, et par le « far plan », représenté par la ligne verticale à droite. Les deux autres lignes formant le cône correspondent au champ de vision :



Le principe de la projection

Un objet qui peut être dessiné à l'écran est caractérisé par un ensemble de sommets, généralement appelés vertex, ou vertices au pluriel. Un vertex est composé à minima d'une position dans l'espace et contient généralement des informations supplémentaires telles que la couleur associée au sommet, une normale, une tangente, des texels, etc.

Ces vertices sont regroupés en primitives : des lignes ou bien des triangles.

Le rendu à l'écran se fait grâce à la carte graphique : on commence par lui passer les vertices à dessiner et pour chacun desquels elle exécutera un vertex shader, un petit programme chargé d'effectuer du traiment qui peut altérer l'état de ce vertex.

Vient ensuite l'étape de la rastérisation, durant laquelle la carte graphique va générer une matrice de pixels qui pourra être affichée sur l'écran. Ces pixels sont envoyés au pixel shader, un petit programme permetttant également le rendu final en y appliquant des effets visuels.

Exercice 1 – Dessiner une forme avec XNA

Dans cet exercice, vous allez appliquer les connaissances théoriques abordées précédemment dans la création d'une application avec XNA.

1. Démarrer Visual Studio 2010 Express pour Windows Phone

   > **Remarque:** toutes les étapes décrites dans cet atelier s'appuient sur la version Express de Visual Studio 2010 pour Windows Phone, mais il est tout à fait possible d'utiliser une version complète de Visual Studio 2010 accompagnée des outils de développement pour Windows Phone.

2. Pour ouvrir Microsoft Visual Phone Developer 2010 Express rendez-vous dans le menu **Démarrer | Tous les programmes | Microsoft Visual Studio 2010 Express**.

   > **Visual Studio 2010:** Pour ouvrir Visual Studio 2010 rendez-vous dans le menu **Démarrer | Tous les programmes| Microsoft Visual Studio 2010**.

3. Dans le menu **File**, cliquez sur **New Project**.

   > **Visual Studio 2010:** Dans le menu File, pointez **New** et cliquez sur **Project**.

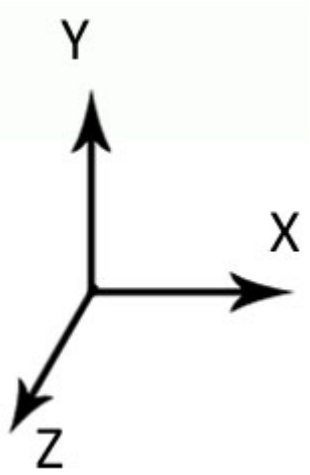
4. Dans la fenêtre **New Project**, choisissez la catégorie **XNA Game Studio 4.0** et selectionnez **Windows Phone Game (4.0)** dans la liste des templates installés. Nommez le projet **Atelier_7** et cliquez sur **OK**.
5. Renommer la classe générée par défaut (**Game1**) en **Atelier_7**. Pour se faire, faites un clic droit sur le nom de la classe et sélectionnez Refactor -> Rename
6. Dans la fenêtre qui s'affiche, entrez Atelier_7 dans le champ **New name** et cliquez sur **OK**.
7. Parcourez les changements que vous propose Visual Studio et cliquez sur le bouton **Apply** pour les appliquer.
8. Supprimez le champ de type **SpriteBatch** de la classe **Atelier_7**.
9. Créez deux champs de type **Matrix** : **projection** et **view**.

```
Matrix projection;
Matrix view;
```

10. Dans la méthode Initialize, commencez par définir la matrice de projection. Les différents paramètres sont ceux expliqués précédemment dans cet atelier.

```
protected override void Initialize()
{
    float aspectRatio = graphics.GraphicsDevice.Viewport.Width / graphics.G
raphicsDevice.Viewport.Height;
    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, as
pectRatio, 0.1f, 10f);
    view = Matrix.CreateLookAt(new Vector3(2, 0, 5), Vector3.Zero, Vector3.
Up);

    base.Initialize();
}
```

Faîtes de même avec la matrice de vue. Notez que sa position est légèrement décalée sur l'axe X et l'axe Y. Notez aussi qu'avec XNA, le repère utilisé est un repère main droite.



Le repère utilisé au sein d'XNA

11. Définissez ensuite la forme à afficher. L'extrait de code suivant représente un triangle. Comme vu précedémment, chaque vertex peut contenir des informations en plus d'une simple position. Ici, c'est le type **VertexPositonColor** qui est utilisé : le vertex contient alors une couleur, la couleur de chaque pixel de l'image finale sera déterminée grâce à une étape d'interpolation.

    Il vous est également possible de créer vos propres types de vertices en implémentant l'interface **IVertexType**.

```
VertexPositionColor[] vertices;


private VertexPositionColor[] CreateTriangle()
{
    VertexPositionColor[] triangle = new VertexPositionColor[3];

    triangle[0] = new VertexPositionColor(new Vector3(-
1, 1, 0), Color.White);
    triangle[1] = new VertexPositionColor(new Vector3(1, -
1, 0), Color.White);
    triangle[2] = new VertexPositionColor(new Vector3(-1, -
1, 0), Color.White);

    return triangle;
}
```

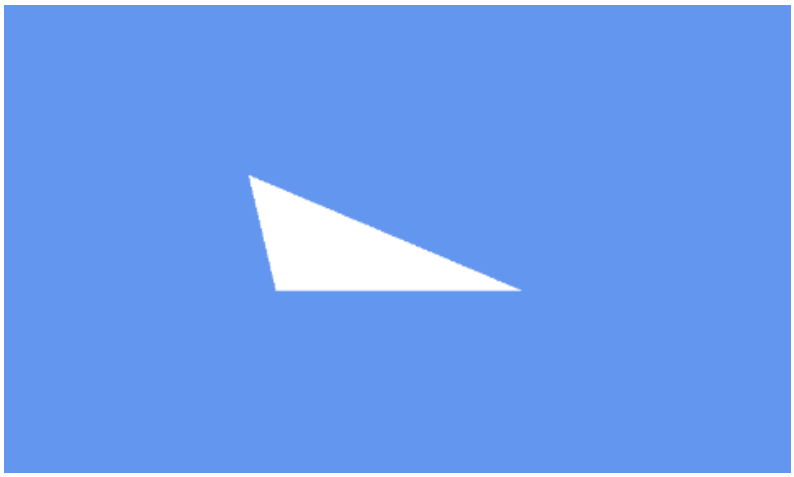
```
protected override void LoadContent()
{
    vertices = CreateTriangle();
}
```

12. Pour afficher la forme à l'écran, il faut qu'elle passe entre les mains d'un vertex shader et d'un pixel shader. Pour se faire, déclarez un champ de type **BasicEffect**. Il existe plusieurs types de d'effets livrés avec XNA, certains permettant d'utiliser des techniques de rendu plus avancées (environnement mapping, dual texture mapping, etc.). Un effet peut contenir plusieurs techniques, elles memes divisées en plusieurs passes. Chacune de ces passes étant reponsable de l'application d'un traitement sur les formes à afficher.

    Il faudra ensuite passer à cet effet les matrices précédemment calculées, puis itérer aux travers de ses passes en communiquant pour chacune d'entres elles le tableau des vertices à afficher.

    Ici, la primitive choisie est telle que les vertices doivent être réunis par trois pour pouvoir former des triangles isolés.

```
BasicEffect basicEffect;

protected override void LoadContent()
{
    vertices = CreateTriangle();
    basicEffect = new BasicEffect(GraphicsDevice);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    basicEffect.View = view;
    basicEffect.Projection = projection;
    basicEffect.World = Matrix.Identity;
    basicEffect.VertexColorEnabled = true;

    foreach (var effectPass in basicEffect.CurrentTechnique.Passes)
    {
        effectPass.Apply();
        GraphicsDevice.DrawUserPrimitives(PrimitiveType.TriangleList, ver
tices, 0, vertices.Length/3);
    }


    base.Draw(gameTime);
}
```



Une première forme en 3D avec XNA

Exercice 2 – Dessin d'une forme plus complexe avec XNA

Dans cet exercice, vous allez apprendre à dessiner des formes plus complexes en utilisant notamment l'index buffer.

Pour dessiner un nouveau triangle collé au premier, il n'est pas nécessaire de déclarer trois nouveaux vertices : il est possible d'utiliser deux des sommets du premier triangle et d'en déclarer uniquement un nouveau. La construction de ces triangles sera alors décrite par ce que l'on appelle l'index buffer.

1. Ajoutez un champs de type **short[]**, il contiendra les indices.

```
short[] indices;
```

2. Modifiez la méthode CreateTriangle comme suit. Les indices qui sont insérés dans l'index buffer correspondent à la position des différents vertices dans leur tableau. Ici, le nouveau triangle se situera en prolongement de l'axe Z et partagera deux sommets avec le premier triangle.

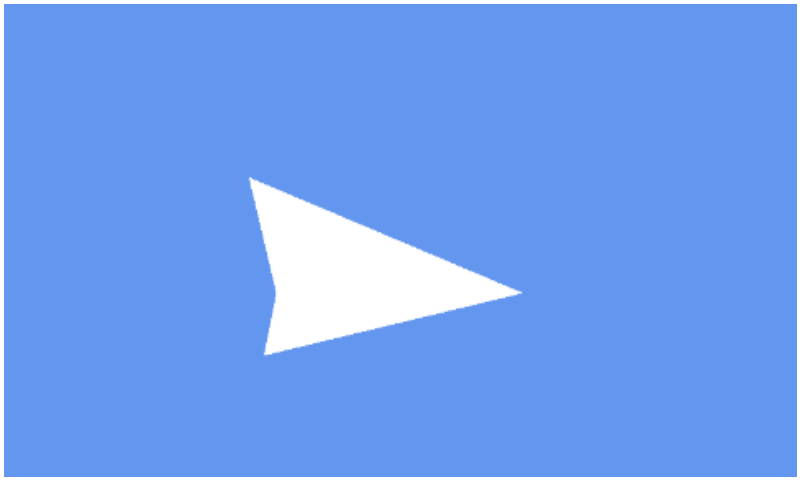
```
private VertexPositionColor[] CreateShape(out short[] indices)
{
    VertexPositionColor[] triangle = new VertexPositionColor[4];

    triangle[0] = new VertexPositionColor(new Vector3(-1, 1, 0), Color.White);
    triangle[1] = new VertexPositionColor(new Vector3(1, -1, 0), Color.White);
    triangle[2] = new VertexPositionColor(new Vector3(-1, -1, 0), Color.White);
    triangle[3] = new VertexPositionColor(new Vector3(-1, -1, 1), Color.White);

    indices = new short[6];

    indices[0] = 0;
    indices[1] = 1;
    indices[2] = 2;

    indices[3] = 1;
    indices[4] = 3;
    indices[5] = 2;

    return triangle;
}
```

3. Modifiez la méthode LoadContent comme ce qui suit.

```
protected override void LoadContent()
{
    vertices = CreateShape(out indices);
    basicEffect = new BasicEffect(GraphicsDevice);
}
```

4. Enfin, modifiez la méthode Draw de manière à ce que l'index buffer soit également passé à la carte graphique. Notez que cette fois ci, le nombre de primitives à dessiner n'est plus déterminé par le nombre de vertices, mais plutôt par le nombre d'indices.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    basicEffect.View = view;
    basicEffect.Projection = projection;
    basicEffect.World = Matrix.Identity;
    basicEffect.VertexColorEnabled = true;

    foreach (var effectPass in basicEffect.CurrentTechnique.Passes)
    {
        effectPass.Apply();
        GraphicsDevice.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, ver
tices, 0, vertices.Length,
                                                 indices, 0, indices.Length/3);
    }

    base.Draw(gameTime);
}
```



Une forme plus complexe grâce à l'index buffer

Exercice 3 – Appliquer une texture à une forme 3D

Dans cet exercice, vous allez découvrir comment utliser un autre type de vertices, ce que sont les coordonnées de texture et comment appliquer une texture à une forme 3D.

1. Commencez par remplacer le tableau de **VertexPositionColor** par un tableau de **VertexPositionTexture**. Cette nouvelle structure ne contient pas d'informations sur la

couleur à appliquer au sommet, mais contient à la place un **Vector2** qui correspond aux coordonnées de texture.
Les coordonnées de texture, ou texels, correspondent à une position dans la texture qui sera mappée au sommet. Cette position peut être exprimée de la manière suivante :

- (0 ; 0) correspond au coin supérieur gauche de la texture ;
- (0 ; 1) correspond au coin inférieur gauche de la texture ;
- (1 ; 1) correspond au coin inférieur droit de la texture.
- (1 ; 0) correspond au coin supérieur droit de la texture ;

En exprimant une valeur supérieure à 1, la texture sera répétée.

Une texture mappée sur un carré sans répétition

Une texture mappée sur un carré avec une repetition sur X

Modifiez le code de votre projet avec l'extrait ci dessous. La forme dessinée ici est un carré, et les texels sont defines de manière à ce que la texture ne soit pas répétée.

```
VertexPositionTexture[] vertices;


private VertexPositionTexture[] CreateShape(out short[] indices)
{
    VertexPositionTexture[] triangle = new VertexPositionTexture[4];

    triangle[0] = new VertexPositionTexture(new Vector3(-
1, 1, 0), new Vector2(0,0));
    triangle[1] = new VertexPositionTexture(new Vector3(1, 1, 0), new Vecto
r2(1, 0));
    triangle[2] = new VertexPositionTexture(new Vector3(1, -
1, 0), new Vector2(1, 1));
    triangle[3] = new VertexPositionTexture(new Vector3(-1, -
1, 0), new Vector2(0, 1));

    indices = new short[6];

    indices[0] = 0;
    indices[1] = 1;
    indices[2] = 2;

    indices[3] = 0;
    indices[4] = 2;
    indices[5] = 3;

    return triangle;
}
```

2. Ajoutez un champ de type **Texture2D**.

```
Texture2D texture;
```

3. Après avoir ajouté une texture au projet, modifiez la méthode **LoadContent** de manière à charger cette texture et la stocker dans le champ que vous venez de créer.

```
protected override void LoadContent()
{
    vertices = CreateShape(out indices);
    basicEffect = new BasicEffect(GraphicsDevice);

    texture = Content.Load<Texture2D>(@"texture");
}
```

4. Modifiez la méthode **Draw** de manière à passer la texture à l'instance de **BasicEffect**, et activez sa propriété **TextureEnabled**.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    basicEffect.View = view;
    basicEffect.Projection = projection;
    basicEffect.World = Matrix.Identity;

    basicEffect.TextureEnabled = true;
    basicEffect.Texture = texture;
```

```
    foreach (var effectPass in basicEffect.CurrentTechnique.Passes)
    {
        effectPass.Apply();
        GraphicsDevice.DrawUserIndexedPrimitives(PrimitiveType.TriangleList
, vertices, 0, vertices.Length,
                                                 indices, 0, indices.Length
/ 3);
    }

    base.Draw(gameTime);
}
```

En executant le projet, le résultat suivant est visible.

5. Modifiez les coordonnées de texture comme ci dessous puis executer l'application.

```
private VertexPositionTexture[] CreateShape(out short[] indices)
{
    VertexPositionTexture[] triangle = new VertexPositionTexture[4];

    triangle[0] = new VertexPositionTexture(new Vector3(-
1, 1, 0), new Vector2(0,0));
    triangle[1] = new VertexPositionTexture(new Vector3(1, 1, 0), new Vecto
r2(1.5f, 0));
    triangle[2] = new VertexPositionTexture(new Vector3(1, -
1, 0), new Vector2(1.5f, 1));
    triangle[3] = new VertexPositionTexture(new Vector3(-1, -
1, 0), new Vector2(0, 1));

    indices = new short[6];

    indices[0] = 0;
    indices[1] = 1;
    indices[2] = 2;

    indices[3] = 0;
    indices[4] = 2;
    indices[5] = 3;

    return triangle;
}
```

Le résultat suivant est visible.

Exercice 4 – Animer un objet

Dans cet exercice, vous allez apprendre à animer un objet en utilisant les classes utilitaires d'XNA, l'objectif étant de le faire tourner autour de l'axe Y à une vitesse d'un quart de tour par seconde.

1. Commencez par ajouter un champ de type **Matrix**. Cette matrice contiendra l'état de notre objet, c'est à dire l'ensemble des transformations qui lui ont été appliqué: translations, rotations et scaling.

```
Matrix world;
```

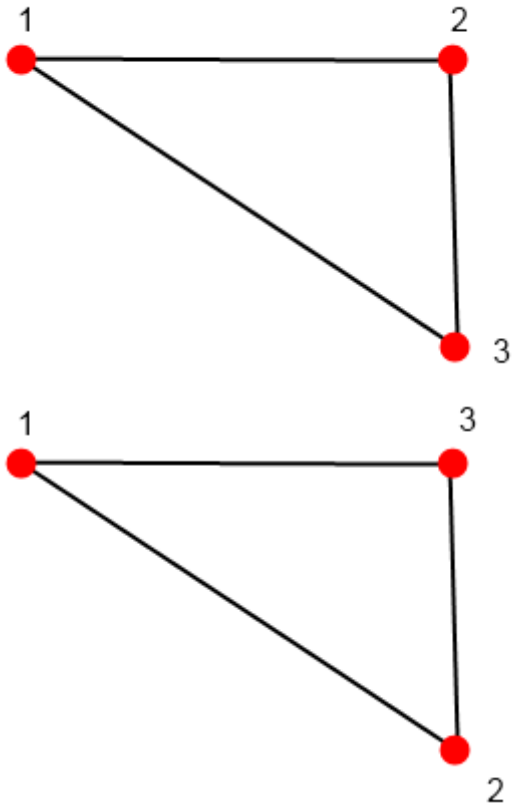
2. Dans la méthode **Update**, vous allez devoir commencer par calculer l'angle de rotation à appliquer en function de la vitesse (90°/s) et du temps écoulé depuis la dernière frame. La classe **MathHelper** fournie une méthode **ToRadians**.
   Vous devrez ensuite utiliser le résultat pour créer une matrice de rotation autour de l'axe Y grâce à la méthode **CreateRotationY**. La structure **Matrix** a l'avantage de nous cacher la relative complexité liée à ces calculs et les spécificités repère main gauche / repère main droite. Elle possède de nombreuses méthodes pour générer des matrices de transformation à la volée, et ses opérateurs mathématiques sont surchargés de manière à pouvoir appliquer facilement ces transformations.

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Press
ed)
        this.Exit();

    world *= Matrix.CreateRotationY((float)(MathHelper.ToRadians(90) * game
Time.ElapsedGameTime.TotalSeconds));

    base.Update(gameTime);
}
```

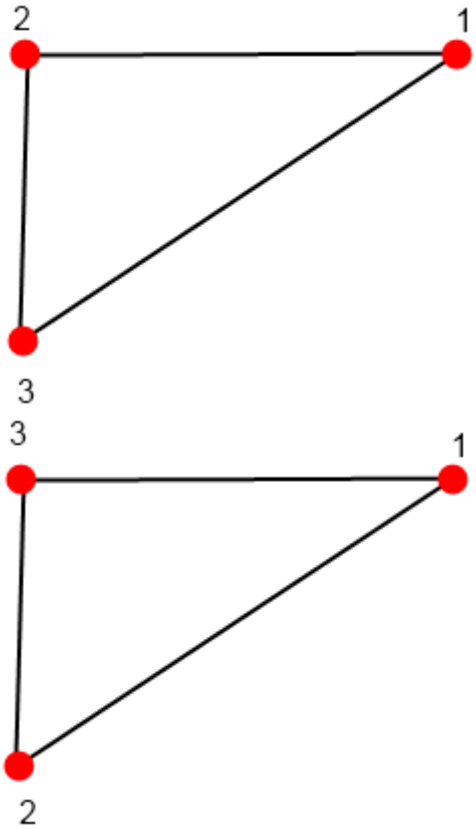
3. Dans la méthode **Draw**, passez simplement cette matrice à l'instance de **BasicEffect**.

```
basicEffect.World = world;
```

4. Executez le projet. L'objet tourne sur lui-même à la bonne vitesse cependant, sa face arrière semble être transparante. Cela s'explique par le fait que par défaut, le backface culling est activé.
   Ce procédé consiste à optimiser les performances en éléminant les faces tournant le dos à la caméra. Cette orientation est déduite en fonction de l'ordre dans lequel sont placés vos vertices lorsqu'ils apparaissent à l'écran.
   Les deux primitives ci-dessous ne sont pas dessinées de la même manière. Pour la première, les indices sont placés tels que les vertices sont dessinés dans le sens horaire et ils le sont dans le sens anti-horaire pour la seconde.



   Par défaut, l'effacement s'applique sur les primitives dessinées dans le sens anti horaire. Lorsque les objets précédents font face à la caméra, le premier est visible mais pas le deuxième. Si on applique une transformation de 180° à ces objets, l'odre dans lequel les sommets sont dessinés est modifié : il devient anti-horaire pour la première primitive et horaire pour la seconde.



La première primitive est alors masquée alors que la seconde devient visible.

Avec XNA, il est possible de modifier les paramètres du **GraphicsDevice** de manière à activer le backface culling pour les faces dessinées dans l'odre horaire ou bien de le désactiver.

Modifiez la fonction **Intialize** de la manière suivante.

```
protected override void Initialize()
{
    float aspectRatio = graphics.GraphicsDevice.Viewport.Width / graphics.G
raphicsDevice.Viewport.Height;
    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, as
pectRatio, 0.1f, 10f);
    view = Matrix.CreateLookAt(new Vector3(0, 5, 5), Vector3.Zero, Vector3.
Up);

    world = Matrix.Identity;

    graphics.GraphicsDevice.RasterizerState = RasterizerState.CullNone;

    base.Initialize();
}
```

Executez à nouveau l'application. Maintenant, l'objet reste visible durant tout son cycle de rotation.

Exercice 5 – Ajouter de la lumière à une scène

Dans cet exercice vous allez apprendre à configurer le **BasicEffect** pour ajouter de la lumière à votre scène et à utiliser un autre type de vertices pour que cette lumière impacte réellement le rendu finale de votre objet.

1. Commencez par remplace le tableau de **VertexPositionTexture** par un tableau de **VertexPositionNormalTexture**. Cette structure contient une information supplémentaire : un **Vector3** correspondant à la normale au sommet. Dans le shader, la quantité de lumièrte à appliquer aux sommets sera calculée en faisant le produit scalaire entre cette normale et la direction de la lumièrte.
   Notez que la méthode **CreateShape** a été modifiée, la forme générée dispose maintenant d'une face arrrière et la direction de ses normales est inversée par rapport à la première face.

```
VertexPositionNormalTexture[] vertices;


private VertexPositionNormalTexture[] CreateShape(out short[] indices)
{
    VertexPositionNormalTexture[] triangle = new VertexPositionNormalText
ure[8];

    triangle[0] = new VertexPositionNormalTexture(new Vector3(-
1, 1, 0), Vector3.Backward, new Vector2(0, 0));
    triangle[1] = new VertexPositionNormalTexture(new Vector3(1, 1, 0), V
ector3.Backward, new Vector2(1.5f, 0));
    triangle[2] = new VertexPositionNormalTexture(new Vector3(1, -
1, 0), Vector3.Backward, new Vector2(1.5f, 1));
    triangle[3] = new VertexPositionNormalTexture(new Vector3(-1, -
1, 0), Vector3.Backward, new Vector2(0, 1));

    triangle[4] = new VertexPositionNormalTexture(new Vector3(1, 1, 0), V
ector3.Forward, new Vector2(0, 0));
    triangle[5] = new VertexPositionNormalTexture(new Vector3(-
1, 1, 0), Vector3.Forward, new Vector2(1.5f, 0));
    triangle[6] = new VertexPositionNormalTexture(new Vector3(-1, -
1, 0), Vector3.Forward, new Vector2(1.5f, 1));
```

```
    triangle[7] = new VertexPositionNormalTexture(new Vector3(1, -
1, 0), Vector3.Forward, new Vector2(0, 1));

    indices = new short[12];

    indices[0] = 0;
    indices[1] = 1;
    indices[2] = 2;

    indices[3] = 0;
    indices[4] = 2;
    indices[5] = 3;

    indices[6] = 4;
    indices[7] = 5;
    indices[8] = 6;

    indices[9] = 4;
    indices[10] = 6;
    indices[11] = 7;

    return triangle;
}
```

2. Modifiez la méthode **Draw** de manière à activer le support d'une lumière directionnelle venant de la droite de la scène et paramétrez sa couleur.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    basicEffect.View = view;
    basicEffect.Projection = projection;
    basicEffect.World = world;

    basicEffect.LightingEnabled = true;
    basicEffect.DirectionalLight0.Enabled = true;
    basicEffect.DirectionalLight0.DiffuseColor = Color.White.ToVector3();
    basicEffect.DirectionalLight0.Direction = Vector3.Left;

    basicEffect.TextureEnabled = true;
    basicEffect.Texture = texture;

    foreach (var effectPass in basicEffect.CurrentTechnique.Passes)
    {
        effectPass.Apply();
        GraphicsDevice.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, vertices,
                                                 indices, 0, indices.Length / 3);
    }

    base.Draw(gameTime);
}
```

Executez le projet. L'objet n'est éclairé que quand une de ses faces est orientée vers la droite.

Exercice 6 – Charger et afficher un modèle 3D

Dans cet exercice, vous allez apprendre à charger un modèle 3D exporté depuis un logiciel de modélisation puis à l'animer.

1. Créez un nouveau projet et ajoutez le modèle tank.fbx dans le Content Project.
2. Avec XNA, les modèles 3D peuvent être chargés au travers de la Content Pipeline via le type **Model**. Un modèle 3D est divisé en bones, sous parties individuellement animables. Animer un modèle consiste à appliquer des transformations à ses bones. Ajoutez donc des références pour chaque bone constituant le modèle ainsi que les matrices contenant les transformations correspondantes.
   N'oubliez pas les classiques matrices de projection & vues pour pouvoir créer le rendu de votre scène.

```
Matrix projection;
Matrix view;
Matrix world;

Model model;
Matrix[] boneTransforms;

ModelBone leftBackWheelBone;
ModelBone rightBackWheelBone;
ModelBone leftFrontWheelBone;
ModelBone rightFrontWheelBone;
ModelBone leftSteerBone;
ModelBone rightSteerBone;
ModelBone turretBone;
ModelBone cannonBone;
ModelBone hatchBone;

Matrix leftBackWheelTransform;
Matrix rightBackWheelTransform;
Matrix leftFrontWheelTransform;
Matrix rightFrontWheelTransform;
Matrix leftSteerTransform;
Matrix rightSteerTransform;
Matrix turretTransform;
Matrix cannonTransform;
```

```
Matrix hatchTransform;

protected override void Initialize()
{
    float aspectRatio = graphics.GraphicsDevice.Viewport.Width / graphics.Gr
aphicsDevice.Viewport.Height;
    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, asp
ectRatio, 1f, 5000f);
    view = Matrix.CreateLookAt(new Vector3(0, 1500, 1500), Vector3.Zero, Vec
tor3.Up);

    world = Matrix.Identity;

    base.Initialize();
}
```

3. Dans la méthode **LoadContent**, chargez votre modèle et initialisez le tableau de matrices en fonction du nombre de bones contenus dans le modèle et récupérez des références sur chacun des bones et sur leurs transformations.

```
protected override void LoadContent()
{
    model = Content.Load<Model>(@"tank");

    boneTransforms= new Matrix[model.Bones.Count];

    leftBackWheelBone = model.Bones["l_back_wheel_geo"];
    rightBackWheelBone = model.Bones["r_back_wheel_geo"];
    leftFrontWheelBone = model.Bones["l_front_wheel_geo"];
    rightFrontWheelBone = model.Bones["r_front_wheel_geo"];
    leftSteerBone = model.Bones["l_steer_geo"];
    rightSteerBone = model.Bones["r_steer_geo"];
    turretBone = model.Bones["turret_geo"];
    cannonBone = model.Bones["canon_geo"];
    hatchBone = model.Bones["hatch_geo"];

    leftBackWheelTransform = leftBackWheelBone.Transform;
    rightBackWheelTransform = rightBackWheelBone.Transform;
    leftFrontWheelTransform = leftFrontWheelBone.Transform;
    rightFrontWheelTransform = rightFrontWheelBone.Transform;
    leftSteerTransform = leftSteerBone.Transform;
    rightSteerTransform = rightSteerBone.Transform;
    turretTransform = turretBone.Transform;
    cannonTransform = cannonBone.Transform;
    hatchTransform = hatchBone.Transform;
}
```

4. Dans la méthode **Update**, appliquez des tranformations à chaque bone en fonction du temps écoulé depuis la dernière frame.

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Presse
d)
        this.Exit();
```

```
    float time = (float)gameTime.TotalGameTime.TotalSeconds;

    leftBackWheelBone.Transform = Matrix.CreateRotationX(time * 5) * leftBac
kWheelTransform;
    rightBackWheelBone.Transform = Matrix.CreateRotationX(time * 5) * rightB
ackWheelTransform;
    leftFrontWheelBone.Transform = Matrix.CreateRotationX(time * 5) * leftFr
ontWheelTransform;
    rightFrontWheelBone.Transform = Matrix.CreateRotationX(time * 5) * right
FrontWheelTransform;
    leftSteerBone.Transform = Matrix.CreateRotationY((float)Math.Sin(time *
0.75f) * 0.5f) * leftSteerTransform;
    rightSteerBone.Transform = Matrix.CreateRotationY((float)Math.Sin(time *
 0.75f) * 0.5f) * rightSteerTransform;
    turretBone.Transform = Matrix.CreateRotationY((float)Math.Sin(time * 0.3
33f) * 1.25f) * turretTransform;
    cannonBone.Transform = Matrix.CreateRotationX((float)Math.Sin(time * 0.2
5f) * 0.333f - 0.333f) * cannonTransform;
    hatchBone.Transform = Matrix.CreateRotationX(MathHelper.Clamp((float)Mat
h.Sin(time * 2) * 2, -1, 0)) * hatchTransform;

    base.Update(gameTime);
}
```

5. Dans la méthode **Draw**, commencez par récupérer les transformations finales de chaque bones relatives à la transformation globale appliquée au modèle via la méthode CopyAbsoluteBoneTransformsTo. Itérez ensuite parmis les différentes parties du modèle puis dessinez les une à une en récupérant leur position dans le tableau de matrices finales.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    model.Root.Transform = world;
    model.CopyAbsoluteBoneTransformsTo(boneTransforms);

    foreach (var mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.World = boneTransforms[mesh.ParentBone.Index];
            effect.Projection = projection;
            effect.View = view;
        }

        mesh.Draw();
    }

    base.Draw(gameTime);
}
```

Exécutez l'application, le tank animé apparait alors sur l'écran de votre téléphone.

Résumé

Cet atelier a introduit les bases du développement d'un jeu ou d'une application en 3 dimensions. En complétant les différents exercices, vous avez pu acquérir les bases pour comprendre de quoi est constituée une scène en 3D, comment l'afficher, et comment afficher des modèles pré conçus à partir de logiciels de modélisation.

Pour aller encore plus loin avec XNA sur Windows Phone 7, vous pouvez vous intéresser à la création de Content Processor pour personnaliser la Content Pipeline et à l'utilisation d'autres effets que le BasicEffect pour améliorer le rendu de vos scènes.