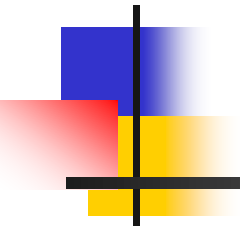
Flex Builder 3

Le 3D en Flex

http://www.flashsandy.org/tutorials/3.0/
http://www.petitpub.com/labs/media/flash/sandy3/

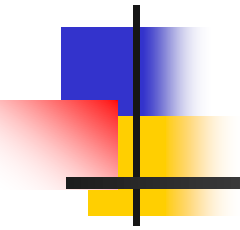


Préambule

- Flex ne permet pas de créer seul la 3D
  - Il passe par la librairie Sandy 3D
    - Créée pour Flash, elle s'adapte à Flex
    - Sandy 3.0.1 est la première version réalisée pour ActionScript3
    - Sources : www.flashSandy.org
  - Elle se charge dans Flex
    - à la création des projets Flex

Préambule

- Le monde de Sandy
  - Sandy affiche une représentation 2D avec des perspectives et des ombres donnant l'illusion d'une 3D
  - Le concept utilisé est proche du monde réel
    - On définit un espace 3D avec
      - des objets 3D dans différentes positions
      - une caméra (œil de l'observateur) regardant la scène à partir d'une certaine position
    - Les objets, ainsi que la caméra, peuvent être déplacés dans l'espace (le monde de Sandy), en créant une projection sur la rétine de l'œil



Préambule

- Grandes étapes de création d’un objet
  - Création de la structure géométrique
    - La structure est attachée à une forme 3D
    - La forme est construite par agencement de polygones (facettes)
  - Création de l'apparence
  - L'apparence est liée à la forme, et automatiquement liée à chaque polygone
    - Ainsi, chaque polygone peut avoir sa propre apparence

Préambule

- Grandes étapes de création d'un objet
  - Une fois la scène créée, on procède à 4 points (pour chaque étape, un événement dédié est diffusé) :
    - Update :
      - on charge les transformations locales
    - Rendering :
      - pour les objets visibles, on procède à leur transformation géométrique
    - Display :
      - chaque « material » (color, bitmap ou vidéo) affiche les polygones avec lesquels ils sont liés

Préambule

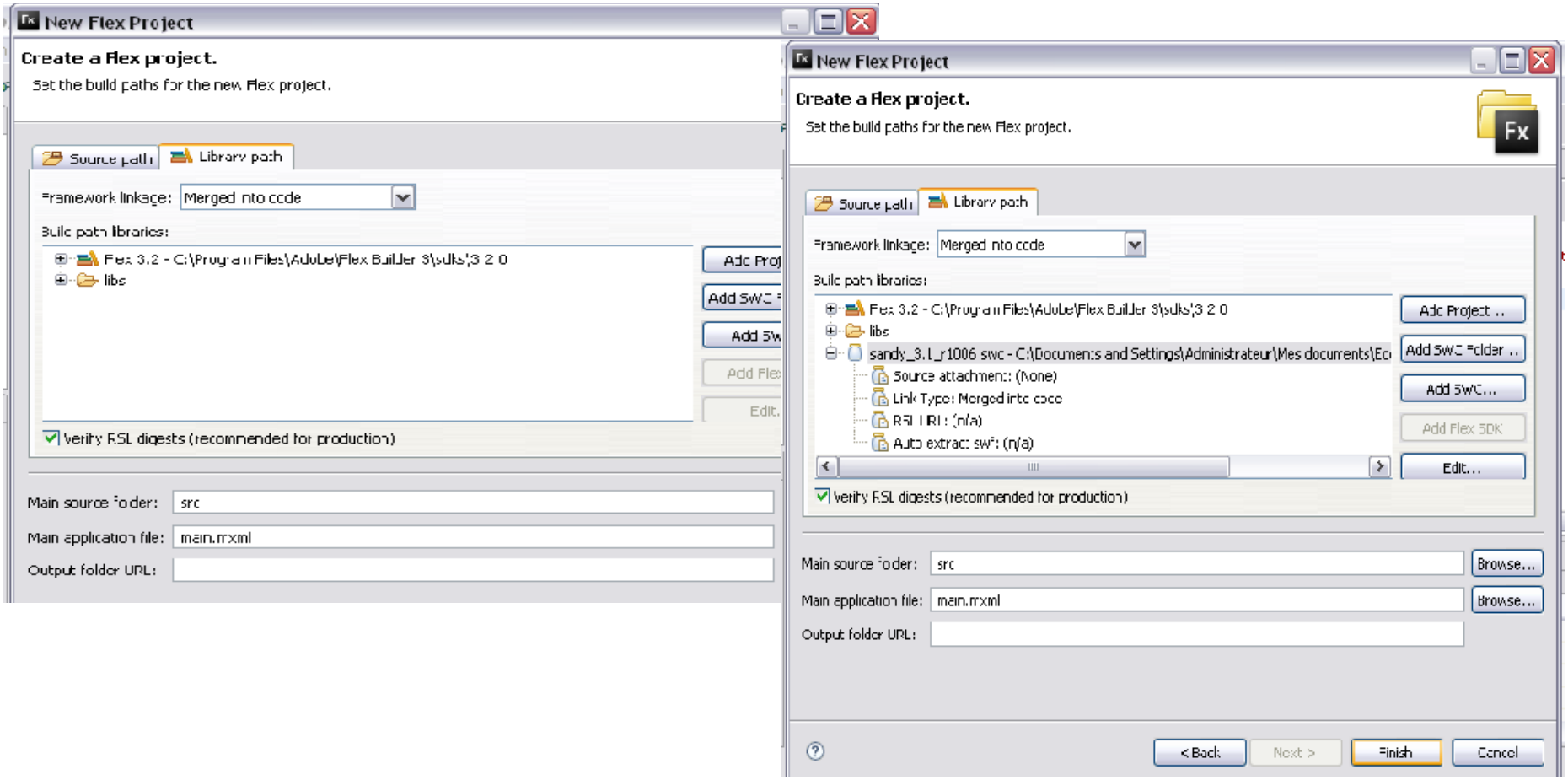
- Performances de Sandy
  - Sandy 3.1 est capable de rendre correctement 2000 à 5000 polygones sur un ordinateur récent, avec une résolution correcte
  - Cela va dépendre de:
    - la taille des textures, et si elles sont animées (une vidéo consommera plus qu'un statique Bitmap),
    - la taille de la « scène », et des objets sur l'écran (plus un objet sera gros, plus il prendra de ressources)
    - la qualité et la puissance du CPU client
    - des réglages de Sandy3D : clipping, containers, lumières... peuvent affecter les performances

Le logiciel Sandy

- Installation de Sandy 3D 3.1
  - Sandy 3D est une librairie externe à Flex. C'est lors de la création d'un projet qu'on va l'importer. La manipulation est simple :

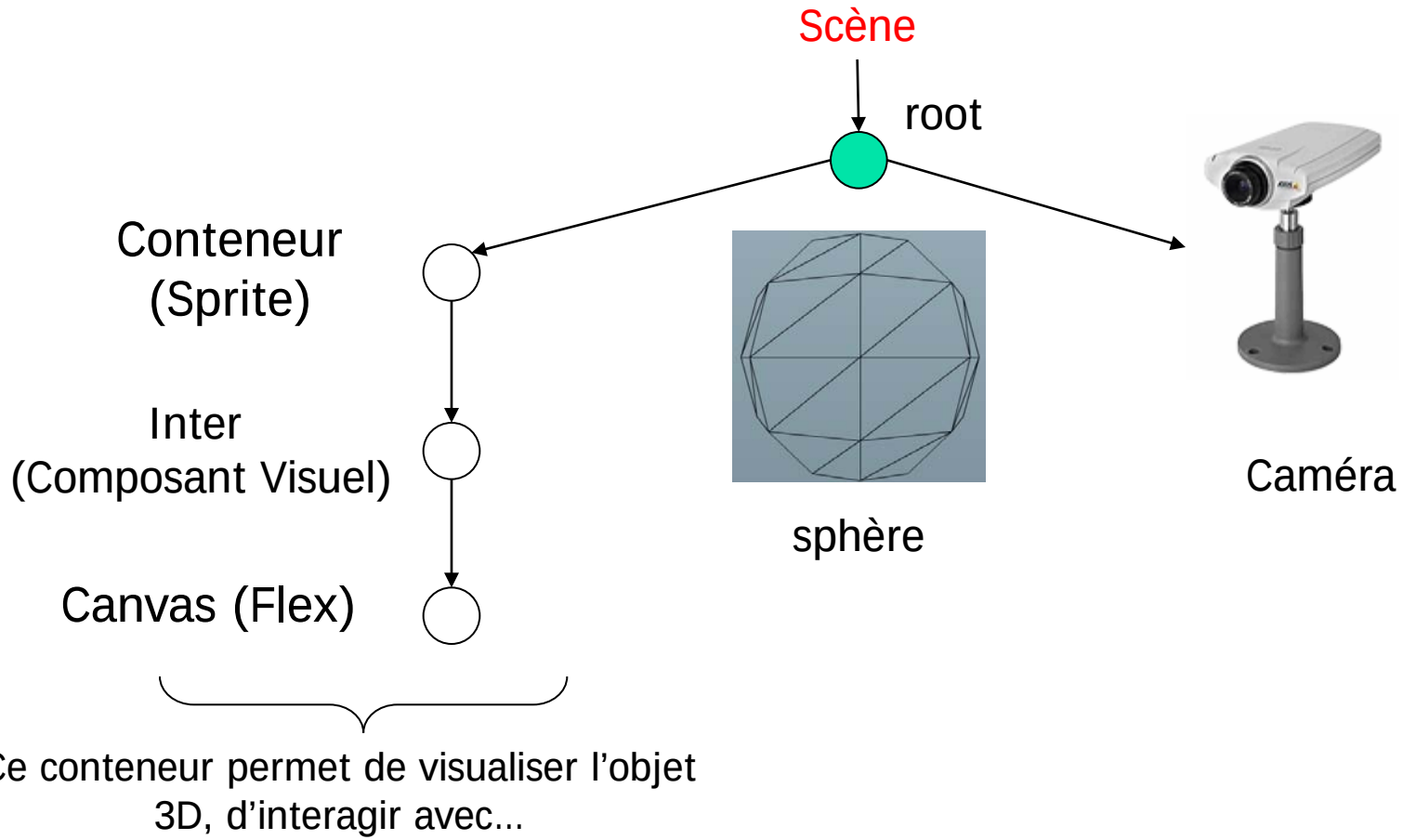
1. Télécharger le fichier sandy_3.1_r1006.swc
2. Après File>New project, choisir un nom puis cliquer sur Next
3. Choisir le dossier de destination, puis refaire « Next »
4. Une page s'ouvre, choisir l'onglet Library Path
5. Dans cet écran, choisir « Add SWC ». Chercher le fichier SWC et l'ajouter au projet
6. Terminer la création du projet par « Finish »

Le logiciel Sandy



Pour créer la scène

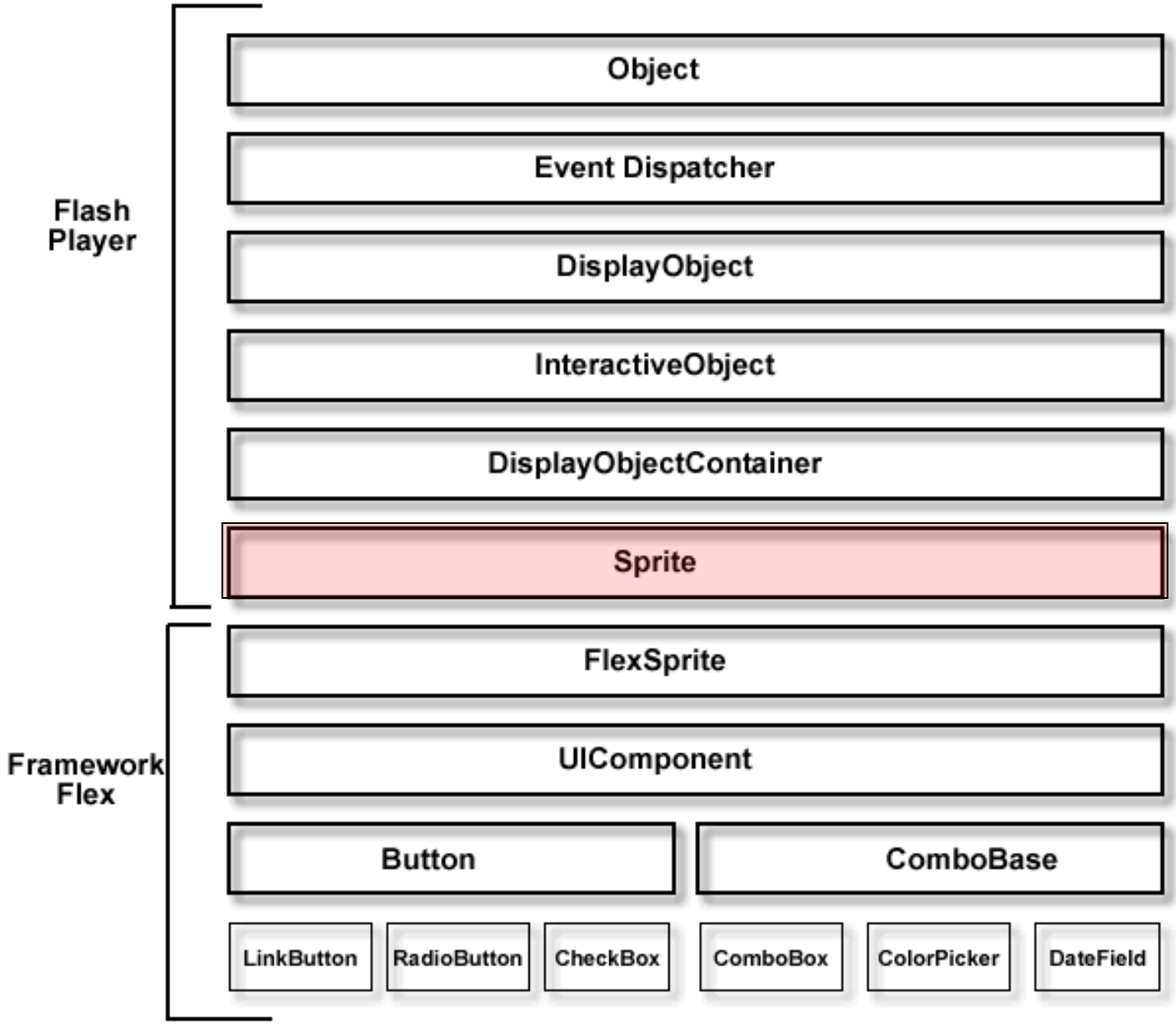
- La scène est formée d'un conteneur de type Flash, d'une caméra et d'une root : un groupe graphique auquel sera attaché l'objet
- Le conteneur Flash permet de visualiser l'objet 3D suivant le point de vue de la caméra



Création de la scène

On utilise les composants visuels de Flex

- Hiérarchie des composants visuels
  - Les composants visuels Flex se comportent comme des objets d'affichage Flash
  - En effet, ceci se retrouve dans la chaîne d'héritage
    - Les composants visuels héritent de : mx.core.UIComponent qui hérite de mx.core.FlexSprite qui hérite directement de flash.display.Sprite qui fait partie de l'API Flash Player


Flash Player
Object
Event Dispatcher
DisplayObject
InteractiveObject
DisplayObjectContainer
Sprite
Framework Flex
FlexSprite
UIComponent
Button
ComboBase
LinkButton
RadioButton
CheckBox
ComboBox
ColorPicker
DateField

Création de la scène

- Prenons un exemple
  - Construire un objet sphère
  - L'exemple suivant montre comment la scène est construite par
    - Niveau 1 : création du conteneur (Sprite)
    - Niveau 2 : création du composant visuel et son ajout au conteneur
    - Niveau 3 : création de la scène par association de la root, de la caméra et du conteneur
    - ajout de l'objet sphère à la scène
    - Définition d'un rendu à la scène

Prenons un exemple : sphère : Simple_ex.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    creationComplete="init()" width="518">
<mx:Script>
<![CDATA[
    import mx.core.UIComponent;
    import sandy.core.Scene3D;
    import sandy.core.scenegraph.*;
    import sandy.primitive.*;

    private var scene:Scene3D;

    private function init():void{
        //Définition d'un composant visuel Flex
        var inter:UIComponent=new UIComponent;

        //Création d'un conteneur Sprite pour y mettre une forme
        var container:Sprite = new Sprite;

        //Ajout du composant visuel à ce container (pour afficher la scène)
        inter.addChild(container);
        MainCanvas.addChild(inter);
```

```
        /* création de la scène. On lui donne ce conteneur, une caméra (un
        point de vue), et un groupe root où ajouter les objets*/
        scene=new Scene3D("maScene", container, new
        Camera3D(500,400), new Group("root"));

        //La caméra est placée au point : (500, 400) avec un recul de 0 par
        rapport à la scène

        // On ajoute un objet à la scène que l'on vient de créer
         scene.root.addChild(new Sphere("maScene"));

        //On donne un rendu à la scène
        scene.render();
        }
]]>
</mx:Script>

//Déclaration du conteneur de la scène
<mx:Canvas id="MainCanvas" width="100%" height="100%">
</mx:Canvas>
</mx:Application>
```

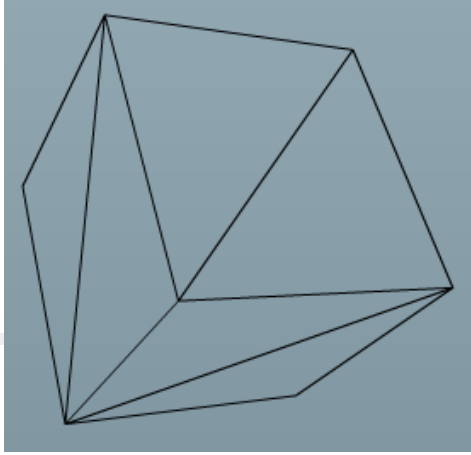
Flex 3D

- Commentaires
  - Les imports
    - Il faut importer les classes de Sandy même si la librairie est ajoutée au projet
  - Les variables importantes : 2 concepts forts de Sandy
    - Scene3D : conteneur principal des objets
    - Camera3D : point de vue de l'utilisateur des objets initiaux

Flex 3D

- Démarche de construction
  - Il faut adopter une stratégie de codage qui reste la même
  - Ici, on suivra le schéma :
    - On crée la Caméra et la Scène,...
    - On place la création de tous les objets dans une fonction privée nommée “init()”
    - Seule la déclaration de la variable Scene sera en dehors
    - On ajoute un EventListener qui va s'occuper de l'initialisation et de l'éventuelle interaction avec l'utilisateur (ce sera fait dans l’exemple suivant)

Flex 3D



- Deuxième exemple
  - Création d'un cube : ex001.mxml
  - On retrouve la même stratégie de construction
  - On reprécise certains éléments, comme :
    - initialisation de la caméra:

      camera = **new** Camera3D(300, 300);

      ➔ position par défaut : (0,0,0)

      camera.z = -400;

      ➔ Régler la caméra à -400 = reculer de 400px de l'origine de la Scène
    - *Ajout d'un EventListener*

      addEventListener(Event.ENTER_FRAME, enterFrameHandler);

      qui dit à la scène de s'initialiser
  - *On tourne le cube pour donner l'illusion de la 3D*

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" creationComplete="init()">
<mx:Script><![CDATA[
    import mx.core.UIComponent;import flash.display.Sprite;
    import flash.events.*;import sandy.core.Scene3D;
    import sandy.core.data.*;import sandy.core.scenegraph.*;
    import sandy.materials.*;import sandy.materials.attributes.*;
    import sandy.primitive.*;

     private var scene:Scene3D; private var camera:Camera3D;

    private function init():void{
     {
    // On crée la caméra
     camera = new Camera3D( 300, 300 );
     camera.z = -400;
     // On crée le "groupe" qui est l'arbre de tous les objets visibles
      var root:Group = createScene();
     // On crée une Scène et on ajoute la caméra et le groupe d'objets
    scene=new Scene3D("maScene", container, new Camera3D(500,400),
    root);
    // On ajoute un écouteur à la scène
      addEventListener( Event.ENTER_FRAME, enterFrameHandler );
   }
```

```
        // On crée le graphisme de la scène basé sur la racine Groupe de la scène
         function createScene():Group
        { // On crée la racine Group
           var g:Group = new Group();

           // On crée un cube
           var box = new Box("box",100,100,100);
            box.rotateX = 30;box.rotateY = 30; // donne l'illusion 3D

          // On ajoute le cube au Groupe
         g.addChild(box);
         return g;
       }
        // L' "Event.ENTER_FRAME" event handler dit à la scène de s'initialiser
        function enterFrameHandler( event : Event ) : void
        {
           scene.render();
         }
}
]]>
</mx:Script>
<mx:UIComponent id="container" width="100%" height="100%"/>
</mx:Application>
```

Explications

- *Déclarations Scene3D et Camera3D*

  ***private var** scene:Scene3D;*

  ***private var** camera:Camera3D;*

  – Dans les déclarations, on peut trouver deux variables de type Scene3D et Camera3D

  – La première représente la « scène » des objets. Il faut la prendre comme étant le conteneur de tout ce que l'on va afficher ou utiliser

  – La seconde est une « caméra » qui va être notre vue sur l'(les) objet(s). Il est donc nécessaire de l'initialiser en un point qui sera le point de vue de l'utilisateur des objets initial :

  camera = new Camera3D(300, 300);

- *Déclarations Scene3D et Camera3D (suite)*
  - Nous devons positionner la caméra :
    - la position par défaut est au point (0,0,0)
    - Régler la caméra à -400 signifie simplement que nous allons nous reculer de 400px de l'origine de la Scene

      camera.z = -400;

- *Création d'un groupe*
  - Nous créons ensuite un objet « Group »

    ```
    var root:Group = createScene();
    ```
  - C'est le groupe de tous les objets dans la Scene
  - Il faut penser aux relations entre nos objets comme « en arbre », où chaque branche de l'arbre peut être un groupe ou un autre objet
  - Le groupe racine (root) est défini dans la fonction **createScene()**, qui sera expliquée plus loin

- *Définition de la scène*
  - On définit la Scene3D

    scene = new Scene3D("scene",container, camera, root);
  - Dans ce cas, on passe 4 variables dans le constructeur :
    - le nom de la scène, le conteneur de la scène, la caméra qui regarde la scène, et enfin la racine de l'arbre des objets

- *Ajout d'un EventListener*
  - On ajoute un eventListener qui va initialiser graphiquement la scène

    addEventListener(Event.ENTER_FRAME,
    enterFrameHandler);
  - Dans ce cas particulier, on n'a pas besoin d'un EventListener tant que l'on ne fait rien, et on aurait pu simplement placer la commande scene.render()
  - Mais, pour bien coder, on met cette commande dans l'Event Handler

- *Objets*
  - On voit maintenant comment construire les objets de la scène
    - Dans le simple exemple du cube, la première chose que l'on fait dans la fonction createScene() est définir l'élément Group qui va être utilisé en paramètre pour l'objet scene3D

      var g:Group = new Group();

- *Construire le cube*
  - On crée le cube de 100 px pour chaque côté

```
var box = new Box( "box",100,100,100);
```

- *Rotation du cube*
  - On tourne le cube de 30 degrés par rapport aux axes X et Y pour créer l'illusion de 3D; sinon, le graphisme du cube aura l'air d'un simple carré.

```
box.rotateX = 30;
box.rotateY = 30;
```

- *Ajouter le cube au groupe*
  - Enfin, il faut ajouter l'objet créé au groupe racine, sinon le moteur de 3D ne saura pas quoi initialiser graphiquement.

```
g.addChild( box );
return g;
```

- *UIComponent*
  - On déclare le conteneur général dans la partie MXML.

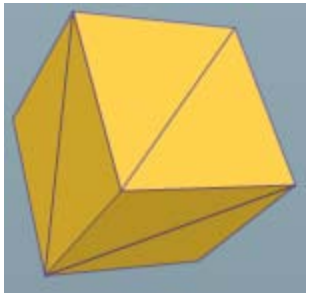
```
<mx:UIComponent id="container" width="100%"
  height="100%"/>
```

Flex 3D

- Notion de Skin (habillage)
  - Concept important dans Sandy (aussi un concept que tous les logiciels de graphisme d'objets ont) :
    - Materials/Appearance (appelés communément « skin »)
  - Avec Sandy, si l'on ne déclare pas d' « appearance » pour un objet qu'on construit, il aura l'air d'un squelette vide (ce que l'on a eu jusqu'à maintenant)
  - Le cube va donc être doté d'une simple couleur, pour commencer, mais on peut aller jusqu'à un support vidéo
    - La version 3 de Sandy compte 9 type de « Materials »

Flex 3D

- Notion de Skin (habillage)
  - Exemple : habiller le cube précédent : ex002.mxml



Flex 3D

Notion de Skin

- Les étapes :
  1. Création de l'attribut matière

```
var materialAttr:MaterialAttributes =
new MaterialAttributes(new LineAttributes( 0.5,
    0x2111BB, 0.4 ), new LightAttributes( true, 0.1))
```

  - **LineAttributes** : précise les *dimensions*, *couleur* et *transparence* des lignes utilisées pour dessiner le cube
  - **LightAttributes** : précise la présence ou non de la lumière et le niveau de lumière

  – Ces attributs doivent implémenter l'interface IAttributes

2. Application de l’attribut material

- Maintenant que l'*attribut* du *material* est défini, on doit spécifier la nature du *material* sur laquelle nous voulons appliquer cet *attribut*
- Dans cet exemple, nous l’appliquerons à ColorMaterial qui est le matériau le plus facile à utiliser

```
var material:Material = new ColorMaterial(
    0xFFCC33, 1, materialAttr );
```

- Le ColorMaterial prend trois paramètres :
  - la couleur des faces du cube
  - la transparence de la couleur
  - et l'attribut du matériel que nous venons de définir

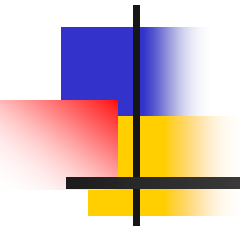
3. Réglage de la lumière pour la matière
   - On demande au moteur de rendu d'utiliser la lumière

     material.lightingEnable = true;

   - ➔ Les couleurs des faces seront initialisées avec des ombres :
     - ➔ face frontale plus lumineuse, tandis que les faces de derrière moins
4. Application (affectation) de l'apparence à la matière

   var app:Appearance = new Appearance(material);
5. Affectation de l'*apparence* à l'objet concerné, ici le cube

   box.appearance = app;



Comprendre la notion de caméra

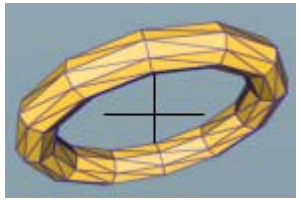
- Notion de point de vue de la scène
  - Jusqu'à présent, on a juste utilisé la camera comme point de vue de la scène entière, mais on ne l'a jamais vraiment utilisée, ni été loin dans ses propriétés ou méthodes
  - Une Camera 3D est plus qu'un objet statique :
    - on peut la bouger autour d'un objet, la tourner...
  - Savoir qu'on peut bouger la caméra est très important car parfois, on peut se demander si il est préférable de bouger les objets autour de la caméra, ou à l'inverse, bouger la caméra autour des objets

Comprendre la notion de caméra

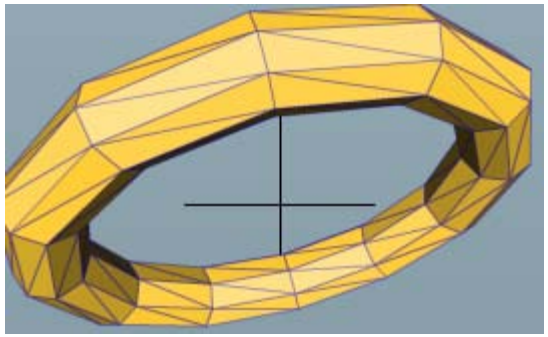
- Le meilleur conseil est :
  - si on veut regarder bouger un seul objet devant la caméra, en laissant les autres dans leur position originale, il vaut mieux bouger l'objet seulement
  - si on veut bouger tous les objets autour de la camera, il vaut mieux bouger uniquement la caméra, et non les objets
- Par exemple
  - si la caméra explore une pièce, on va bouger la caméra et non tous les murs...
  - on va laisser l'exemple du cube et utiliser d'autres primitives : Line3D et un objet « Torus »

Comprendre la notion de caméra

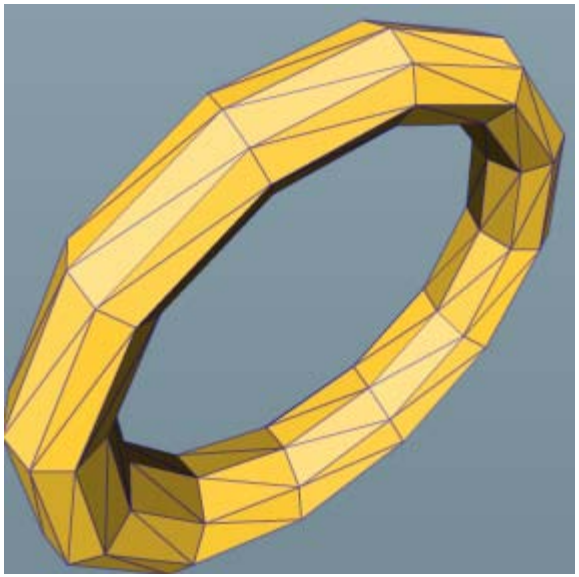
- Exemple : ex003.mxml
  - Regardez le code



Éloignement



Rapprochement



Positionnement de
la caméra à droite

Comprendre la notion de caméra

- *Commentaires*
  - *La création de la scène 3D se fait dans* createScene(),
    - débute par la création d'un système de coordonnées avec 3 axes

      var myXLine:Line3D = **new** Line3D(**"x-coord"**,
      new Point3D(-50, 0, 0), **new** Point3D(50, 0, 0));

      …
    - crée ensuite l'objet tore

      var torus:Torus = new Torus("theTorus", 120, 20);
      - *3 paramètres : le nom, le rayon de l'anneau et le rayon de l'anneau central*
    - définit la matière et l'apparence
    - rattache le système de coordonnées et l'objet tore au groupe

Comprendre la notion de caméra

- La gestion de l'événement
  - Consiste à déplacer, tourner le tore en utilisant les touches clavier
  - *On ajoute deux écouteurs d'événements pour faire des déplacements de caméra et observer suivant différents points de vue le Tore*
  - Le premier écouteur est responsable du rendu de toute la scène

    addEventListener0; Event.ENTER_FRAME, enterFrameHandler);
  - Le deuxième intercepte les événements saisis au clavier

    stage.addEventListener0; KeyboardEvent.KEY_DOWN, keyPressed);

Détails

- *Les imports:*
  - Il y a de nouveaux imports :

```
import mx.core.UIComponent;
import flash.display.Sprite;
import flash.events.*;
import flash.ui.*;
import sandy.core.Scene3D;
import sandy.core.data.*;
import sandy.core.scenegraph.*;
import sandy.materials.*;
import sandy.materials.attributes.*;
import sandy.primitive.*;
```

- *Redéfinition de la position de la Camera*
  - On a ajouté trois lignes, en commentaires

```
camera = new Camera3D( 300, 300 );
//camera.x = 100;
//camera.y = 100;
camera.z = -400;
//camera.lookAt(0,0,0);


//camera.x = 100; // Bouge la caméra vers la droite
//camera.y = 100; // Bouge la caméra vers le haut
//camera.lookAt(0,0,0); // Définit la direction vers
   laquelle regarder
```

- *Redéfinition de la position de la Camera (suite)*
  - Dans l'exemple, on n'utilise pas ces réglages
  - Vous pouvez enlever les commentaires pour voir l'effet des instructions, mais le but est de comprendre que l'on peut bouger la caméra où l'on veut en utilisant les propriétés de x, y, a
  - On peut aussi préciser à la caméra vers où elle doit pointer dans la scène, en utilisant camera.lookAt()

- *Placer un axe de référence*
  - Dans la fonction createScene() , il y a trois lignes ajoutées, qui représentent les trois axes majeurs du notre système

```
// on crée un système de coordonnées
var myXLine:Line3D = new Line3D( "x-coord", new
   Point3D(-50, 0, 0), new Point3D( 50, 0, 0 ));
var myYLine:Line3D = new Line3D( "y-coord", new
   Point3D(0, -50, 0), new Point3D( 0, 50, 0 ));
var myZLine:Line3D = new Line3D( "z-coord", new
   Point3D(0, 0, -50), new Point3D( 0, 0, 50 ));
```

- *Placer un Torus*
  - On utilise une nouvelle primitive : le *torus*

    ```
    var torus:Torus = new Torus( "theTorus", 120, 20);
    ```
  - Le constructeur prend trois paramètres : le nom, le rayon de l'anneau , et le rayon de l'anneau central (de combien le tube sera épais).

- *Ajouter des objets au groupe*
  - On ajoute ces quatre objets au groupe racine.

```
g.addChild(myXLine);
g.addChild(myYLine);
g.addChild(myZLine);
g.addChild( torus);
```

- *Construction de l'EventListener*
  - Un nouvel EventListener a été ajouté.

    addEventListener(Event.ENTER_FRAME, enterFrameHandler);

    stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);
  - Le premier écouteur est responsable de l'affichage graphique de la scène entière, tandis que les autres sont chargés d'écouter les actions de l'utilisateur :
    - dans ce cas KeyboardEvent

- Regardons ce que fait la méthode keyPressed :

```
switch(event.keyCode) {
    case Keyboard.UP: camera.tilt +=2; break;
    case Keyboard.DOWN: camera.tilt -=2; break;
    case Keyboard.RIGHT: camera.pan -=2; break;
    case Keyboard.LEFT: camera.pan +=2; break;
    case Keyboard.CONTROL: camera.roll +=2; break;
    case Keyboard.PAGE_DOWN: camera.z -=5; break;
    case Keyboard.PAGE_UP: camera.z +=5; break;
}
```

- La méthode keyPressed agit différemment selon la touche préssée
- Elle utilise seulement des propriétés de la classe Camera3D

- *Le résultat*
  - Pour que la méthode keyPressed fonctionne, il faut cliquer dans la scene
  - Ensuite, on peut bouger la camera dans chaque direction en utilisant les touches haut, bas, droit ou gauche
  - Avec les touches haut de page, bas de page, on peut zoomer
  - On peut tourner la caméra avec la trouche Control.
  - On bouge la caméra et non l'objet; l'objet est stationnaire, comme on peut le voir, sa position est constante par rapport au système coordiné

- Pour s'en rendre compte, on peut modifier l'exemple 3 de façon à ce que :
  - Les positions camera.x et camera.y soient 0 (le défaut), dans un premier fichier.
  - Dans un second, camera.x and camera.y asont réglés à 100, on peut ainsi voir l'axe Z
  - La caméra regarde « tout droit »
  - Dans un autre fichier, la caméra regarde le point (0,0,0) qui est le centre des axes
- Jouer avec les exemples pour se fixer une idée de comment gérer la caméra

Comprendre la notion de caméra

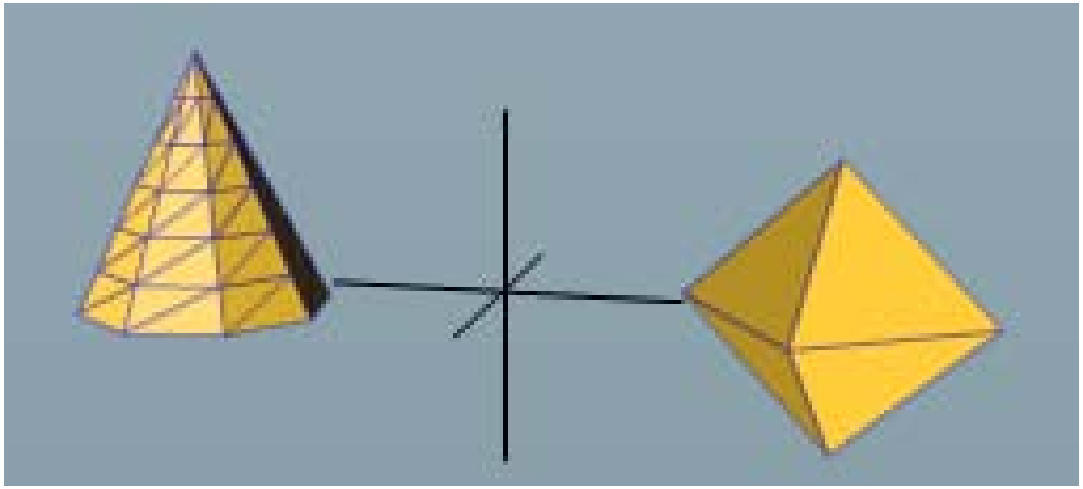
- *Bouger des objets autour de la scène 3D*
  - Nous savons donner l'effet que les objets tournent autour de la caméra en bougeant en fait la caméra
  - Mais comment les faire bouger ?
  - On peut bouger un objet à l'écran, ou bien un groupe contenant un ou plusieurs objets
  - Dans le but de bouger un seul objet, une fois que l'on a fait référence à lui, on peut le faire tourner autour des trois axes de référence en utilisant les propriétés **rotateX**, **rotateY**, et **rotateZ**
  - On utilise les propriétés **pan**, **roll**, et **tilt** pour faire tourner l'objet autour de ses axes locaux

Comprendre la notion de caméra

- *Bouger des objets autour de la scène 3D (suite)*
  - Si on veut bouger un groupe d'objets, on doit introduire une nouvelle classe, nommée **TransformGroupn,** qui implémente l'interface **Atransformable**, et on pourra le bouger comme un simple objet
  - On va, pour l'expliquer, dessiner deux objets
    - On verra la différence entre bouger un simple objet et le groupe entier

Bouger des objets autour de la caméra

- Exemple d'illustration : ex004.mxml
  - Ici, nous allons dessiner 2 objets et montrer comment bouger un objet et bouger un groupe d'objets



Bouger des objets autour de la caméra

- Comment faire ? Examinons le code
  1. *Déclaration des objets nécessaires*
     - D'abord, on doit définir les variables objets en privé

```
// le groupe transformable
private var tg:TransformGroup;
// le cône
private var myCone:Cone;
// le tétraèdre
private var myHedra:Hedra;
```

     - Cela nous permet de nous référer à ces variables depuis partout dans le code

2. *Création du TransformGroup*
   - Dans la méthode createScene(), on crée l'objet tg par instanciation de la classe TransformGroup

     tg = new TransformGroup('myGroup');
     - TransformGroup apportera deux nouveaux éléments : un Cône et un tétradèdre
3. Création et positionnement des objets
   - Nous créons ensuite les deux nouveaux objets et on les positionne correctement pour qu'ils ne se chevauchent pas

     myCone = new Cone("theObj1",50, 100);

     myHedra = new Hedra("theObj2", 80, 60, 60);

     myCone.x = -160;

     myCone.z = 150;

     myHedra.x = 90;

4. On peut ajouter une apparence pour chaque objet

```
myCone.appearance = app;
myHedra.appearance = app;
```

5. On les ajoute à la transformGroup
   - Ainsi, nous aurons une référence à un «nœud» que l'on peut déplacer comme on veut

```
tg.addChild(myCone);
tg.addChild(myHedra);
```

6. On ajoute les objets restant au groupe principal

```
g.addChild(tg);
g.addChild(myXLine);
g.addChild(myYLine);
g.addChild(myZLine);
```

- Remarque
  - Attention, on doit juste ajouter :
    - le TransformGroup que l'on veint de créer et les trois lignes
    - On n'ajoute pas les deux autres objets puisqu’on les a déjà ajoutés au TransformGroup

7. Gérer les événements
   - Voyons voir quels événements nous gérons ?
   - Dans la fonction enterFrameHandler, on a décidé de montrer comment vous pouvez accéder aux objets simples, directement
   - Ainsi, nous pouvons faire pivoter chaque objet, à titre individuel

```
myHedra.pan +=4;
myCone.pan +=4;
scene.render();
```

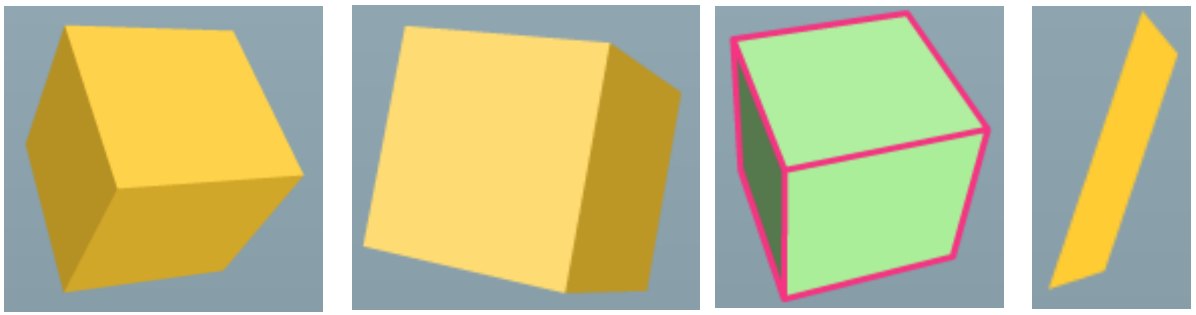
8. Gérer les événements
   - Dans la fonction keyPressed, on a voulu montrer comment accéder à l'ensemble du Groupe
   - Ainsi, vous pourrez voir le mouvement de chacun des deux objets ensemble

```
switch(event.keyCode) {
    case Keyboard.UP: tg.y +=2; break;
    case Keyboard.DOWN: tg.y -=2; break;
    case Keyboard.RIGHT: tg.roll +=2; break;
    case Keyboard.LEFT: tg.roll -=2; break;
}
```

9. Et maintenant tester :
   - Appuyez sur les flèches du clavier pour faire tourner l'ensemble du groupe
   - Notez que les trois axes ne se déplacent pas, puisque nous ne vous déplacez pas la caméra. Et les axes ne sont pas à l'intérieur du TransformGroup

Bouger des objets autour de la caméra

- Bouger l'objet par rapport à la caméra et agir sur son apparence : skin_08.mxml



  - Nous avons déjà vu précédemment comment appliquer une matière simple à un cube
  - Nous allons commencer à partir de là et nous passerons à des effets plus sophistiqués, en agissant sur les polygones du cube

Bouger des objets autour de la caméra

- Le code
  - Vous avez dû remarqué qu'on a utilisé trois variables d'apparence
    - private var app01:Appearance;
    - private var app02:Appearance;
    - private var app03:Appearance;

    car nous voulons utiliser les touches clavier pour les changer de manière interactive
  - Pour chaque attribut, appi, on utilise une matière particulière, ex pour la première :

```
var material01:Material = new ColorMaterial( 0xFFCC33, 1,
    materialAttr01 );
material01.lightingEnable = true;
```

Bouger des objets autour de la caméra

- Avec le deuxième type d'apparence, on définit en plus des lightAttributes, des LineAttribues
- var materialAttr02:MaterialAttributes = new MaterialAttributes(new LightAttributes(true, 0.1), new LineAttributes(3, 0xF43582, 1));
- Les LineAttribues montrent comment sont rendues les lignes qui composent notre modèle
- Le nombre de lignes qui construisent nos modèles repose sur le mode que vous avez choisi, ici 4 car le mode est quad comme quadrilatère
- LineAttribues a 3 paramètres:
  - l'épaisseur de la ligne
  - sa couleur
  - et la valeur alpha en % de l'opacité complète

Bouger des objets autour de la caméra

- Enfin, avec la 3ème apparence, on définit en plus ce qu'on appelle : OutlineAttributes

```
var materialAttr03:MaterialAttributes = new
MaterialAttributes( new LightAttributes( true, 0.1), new
OutlineAttributes(3, 0xFC5858, 1), new LineAttributes(1,
0x000000, 1) );
```

- Ceci permet de définir l'apparence du périmètre de l'objet : c'est la ligne qui entoure l'objet
- On peut également donner une apparence particulière à chaque face du cube

```
box.aPolygons[0].appearance = app01;
box.aPolygons[1].appearance = app02;
box.aPolygons[2].appearance = app03;
box.aPolygons[3].appearance = app02;
box.aPolygons[4].appearance = app01;
box.aPolygons[5].appearance = app03;
```

Rotation d'un objet

- Idée
  - Faire tourner les objets et montrer des faces cachées
  - Trois méthodes
    1. Utiliser un objet d'une classe existante ; cube, sphère, etc. où les faces peuvent être retrouvées facilement à partir de la définition de l'objet
    2. Utiliser un gif animé dans lequel vous avez préparé les 360 vues suivant un axe de rotation ➔ Sprite 3
    3. Utiliser un fichier au format 3DS MAX qui contient toutes les faces de l'objet

Utilisation d’un gif animé

- Possible avec Sprite 3
  - Avant, on travaillait avec Sprite 2D où une seule image était suffisante
  - Pour travailler avec plusieurs images, il faut utiliser Sprite 3D
    - Ici, on doit disposer d’un fichier SWF qui est une séquence d’images pré-rendues
  - L'objet Sprite3D n'est rien d'autre qu'un fichier swf simple, avec 360 images
  - Donc ce que nous devons faire en premier est de choisir un objet qu’on tranche en 360 images (au maximum), afin d'avoir les primitives pour créer notre fichier SWF
  - Le répertoire plane contient les 360 images obtenues par un créateur de gif animé

Introduction à Sprite 3D

- Voici un exemple d'un gif animé qui permet de voir ces 360 images

Introduction à Sprite 3D

- Création de l'application : ex0043.mxml, Les étapes
  1. Chargement du swf
     - on utilise une fonction pratique de sandy : Load

```
queue = new LoaderQueue();
queue.add( "plane", new
    URLRequest("plane/plane.swf") );
queue.addEventListener(SandyEvent.QUEUE_COM
    PLETE, loadComplete );
queue.start();
```

  2. Pour utiliser l'élément importé, juste une ligne de code :

```
s = new Sprite3D("plane",queue.data["plane"],2);
```

     - Le nombre « 2 » est utilisé pour doubler la taille du fichier swf
     - On peut changer ce paramètre du constructeur pour redimensionner des images facilement

http://snakeeyes57.free.fr/fac/Master/M1/Interface%20Hom
me-Machine/Cours8-Exemples/src/

Introduction à Sprite 3D

3. *Gestion de l'objet créé avec le swf*
   - La fonction keyPressedHandler permet de changer la direction de l'objet avec les touches DROITE et GAUCHE

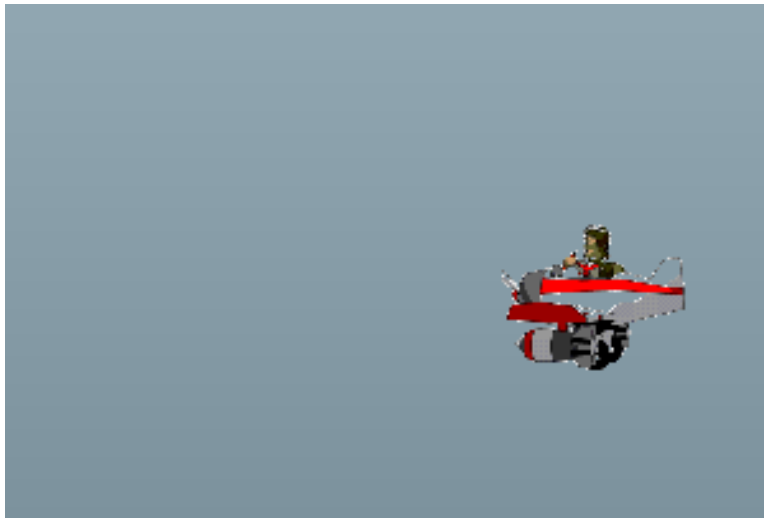
```
private function
    keyPressedHandler(event:KeyboardEvent):void {
     if(event.keyCode == Keyboard.RIGHT)
            s.rotateY -=5;
     if(event.keyCode == Keyboard.LEFT)
            s.rotateY +=5;
     }
```

Introduction à Sprite 3D

3. *Gestion de l'objet créé avec le swf (suite)*
   - On a ensuite besoin d'une fonction qui empêche l'objet de disparaître, qu'on a placé dans l'enterFrameHandler
   - L'objet réapparaîtra sur la gauche s'il va trop loin

```
private function enterFrameHandler( event : Event ) :
    void
    {  if (s.rotateY==0) s.rotateY=0.1;
            if(s.x > 220 && s.z < 0)  s.x=-220;
            else if(s.x < -220 && s.z < 0) s.x=220;
            else if (s.z<-250) s.z=1000;
           s.moveForward(-7);
           scene.render();
    }
```

Introduction à Sprite 3D



Importation de modèles

- Objectif :
  - Importer des fichiers contenant plus de facettes pour faire tourner et bouger l'objet
    - Sandy peut importer des fichiers au format 3DS MAX, et Collada (format non standard)
  - En réalité, Il y a deux façons pour utiliser un modèle 3D provenant d'un autre logiciel :
    - utiliser un modèle qui a été exporté dans un format AS
    - importer un fichier 3DS : vous verrez qu'il faudra un peu plus de code à écrire

Importation de modèles

- Méthode 1 : importing3DS.mxml
  - Le modèle est rangé dans Tereira.as. Vous pouvez jeter un coup d'œil dessus
  - Dans l'application Flex, on crée une variable pot en instanciant la classe Teiera
    - private var scene:Scene3D;
    - private var camera:Camera3D;
    - private var pot:Teiera;
    - pot = new Teiera("pot");
  - Après, on l'utilise comme tout autre objet

Importation de modèles

- Méthode 2 : importing3DS_1.mxml
  - Le modèle est rangé dans teieraASE.ASE
  - La principale différence avec ce qui précède, est qu'il faut importer un parser de Sandy

    import sandy.parser.*;
  - Après, on définit un objet de type Shape3D pour pouvoir interagir avec
    - private var pot:Shape3D;
  - Ensuite vient la partie importante qui importe le fichier 3DS. On essaie de prévenir les erreurs qui peuvent provenir lors de l'import
    - var parser:IParser = Parser.create("asset/teieraASE.ASE",Parser.ASE);
    - parser.addEventListener(ParserEvent.FAIL, onError);
    - parser.addEventListener(ParserEvent.INIT, createScene);
    - parser.parse();

Importation de modèles

- Un autre exemple d'import d'un format 3DMAX : importing3DS_2.mxml
  - Ici, nous apprenons à utiliser la classe **ParserStack** qui ressemble à la classe déjà vue **LoaderQueue**
  - La classe LoaderQueue permet de charger certains visuel assets (images, swf files) tandis que ParserStack nous permet de charger différents modèles 3D basés sur le parsing de fichiers 3D

Importation de modèles

- Un autre exemple d'import d'un format 3DMAX : importing3DS_2.mxml
  - Tout d'abord nous avons besoin de définir les éléments IParser, un pour chaque modèle que nous voulons charger : 4 roues et 1 chassis

```
var parser:IParser = Parser.create("assets/models/ASE/car.ASE",Parser.ASE ); var parserLF:IParser = Parser.create("assets/models/ASE/wheel_Front_L.ASE",Parser.ASE ); var parserRF:IParser = Parser.create("assets/models/ASE/wheel_Front_R.ASE",Parser.ASE ); var parserLR:IParser = Parser.create("assets/models/ASE/wheel_Rear_L.ASE",Parser.ASE ); var parserRR:IParser = Parser.create("assets/models/ASE/wheel_Rear_R.ASE",Parser.ASE );
```

Importation de modèles

- Un autre exemple d'import d'un format 3DMAX : importing3DS_2.mxml
  - Une fois que nous avons créé ces IParser, nous les ajouterons à notre classe PareserStack:
    - parserStack.add("carParser",parser);
    - parserStack.add("wheelLFParser",parserLF);
    - parserStack.add("wheelRFParser",parserRF);
    - parserStack.add("wheelLRParser",parserLR);
    - parserStack.add("wheelRRParser",parserRR);

Importation de modèles

- Dans la méthode parserComplete (..), on extrait les objets Shape3D que nous venons de charger
- Pour ce faire nous allons utiliser une méthode publique fournie par la classe ParserStack: (..): getGroupByName
  - car =parserStack.getGroupByName("carParser").children[0] as Shape3D;
  - wheelLF = parserStack.getGroupByName("wheelLFParser").children[0] as Shape3D;
  - wheelRF =
  - parserStack.getGroupByName("wheelRFParser").children[0] as Shape3D;
  - wheelLR =
  - parserStack.getGroupByName("wheelLRParser").children[0] as Shape3D;
  - wheelRR =
  - parserStack.getGroupByName("wheelRRParser").children[0] as Shape3D;

Importation de modèles

- Maintenant que tous les objets dont nous avons besoin sont créés nous avons besoin de charger les skins pour nos Camero
- Pour ce faire nous allons utiliser la classe LoaderQueue qui se charge d'affecter deux peaux différentes : une pour les roues et un pour le châssis
- var material:BitmapMaterial = new BitmapMaterial(queue.data["carSkin"].bitmapData); … var materialW:BitmapMaterial = new BitmapMaterial(queue.data["wheels"].bitmapData);
- Ces deux materiaux seront utilisés comme entrées pour l'apparence du chassis et des roues pour notre caméra

```
tg = new TransformGroup('myGroup');
…
tg.addChild( wheelLF );
tg.addChild( wheelRF );
tg.addChild( wheelLR );
tg.addChild( wheelRR );
tg.addChild( car );
```

Importation de modèles

- Autre exemple : importing3DS_074.mxml

Importation de modèles

- Autre exemple : sprite2D_forest.mxml