```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace KinemaExample
{
   static class Program
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
         using (MainForm fenetrePrinc = new MainForm())
         {
            //System.Windows.Forms.Application.Idle += new EventHandler(fenetreP
rinc.OnApplicationIdle);
            fenetrePrinc.Show();
            Application.EnableVisualStyles();
            Application.Run(fenetrePrinc);
         }

         /*using (MainForm fenetrePrinc = new MainForm())
         {
            fenetrePrinc.Show();

            // Tant que la form est valide on met à jour le moteur et on traite
les messages
            while (fenetrePrinc.Created)
            {
               fenetrePrinc.UpdateEngine();
               Application.DoEvents();
            }
         }*/

      }
   }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
//using System.Runtime.InteropServices;
using Kinema;

namespace KinemaExample
{
   public partial class MainForm : Form
   {
      private KinemaEngine moteur = null;

      public MainForm()
      {
         InitializeComponent();

         // On s'assure que tout le dessin soit réalisé dans le PaintEvent
         this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque
, true);

         // On force la fenetre à une taille standard
         this.ClientSize = new Size(800, 600);
         this.Text = "Kinema en action";

         // On crée le moteur
         moteur = new KinemaEngine(this);
         //moteur.IsPaused = false;

         moteur.IsDrawingStats = true;
         //moteur.Camera = new Camera(moteur, this, new Vector3(0, 0, 0), new Ve
ctor3(50, 0, 0), new Vector3(0, 1, 0));
         UserControledCamera camera = new UserControledCamera(moteur, this, new
Vector3(0, 0, 0), new Vector3(50, 0, 0), new Vector3(0, 1, 0), (float)Math.PI /
4, 1.0f, 200.0f);
         camera.SetDefaultKeyMapping(UserControledCamera.DefaultMapping.Arrows);
         camera.LookAtCondition = UserControledCamera.MoveLookAtCondition.Mouse1
;
         moteur.Camera = camera;
         /*KinemaMeshObject model = new KinemaMeshObject(moteur, new Vector3(50,
 0, 0), new Quaternion(1, 0, 0, 0), Vector3.Empty, new Vector3(2.0f, -1.0f, 0f),
 "Dragon.x");
         model.Scaling = 0.1f;
         model.GenerateOctree(5, 2);
         model.BoundingSphereTreeLevelDrawing = -1;*/
         /*KinemaMeshObject model = new KinemaMeshObject(moteur, new Vector3(50,
 0, 0), Quaternion.RotationYawPitchRoll(0, 0, 0), new Vector3(0, 0, 0), new Vect
or3(2.0f, 2.0f, 0f), "tiger.x");
         model.Scaling = 5f;
         model.GenerateOctree(6, 5);
         //model.GenerateBinaryTree();
         model.BoundingSphereTreeLevelDrawing = 6;*/
         //model.Torques = new Vector3(0, 5, 0);
         /*KinemaMeshObject model = new KinemaMeshObject(moteur, new Vector3(50,
 0, 0), Quaternion.RotationYawPitchRoll(0f,1.5f,1.5f), Vector3.Empty, new Vector
3(0f, 0f, 0f), "triangle.x");
         model.Scaling = 0.1f;
         model.GenerateOctree(5, 1);
         model.BoundingSphereTreeLevelDrawing = 1;*/
         //moteur.GameObjects.Add(model);
```

```
        //Mesh essai = Mesh.Box(moteur.Device, 2f, 2f, 2f);
        //Mesh essai = Mesh.Sphere(moteur.Device, 1f, 20, 20);
        //essai.WeldVertices(WeldEpsilonsFlags.WeldAll, new WeldEpsilons(), nul
l);
        /*BoundingSphereTree tree = new BoundingSphereTree();
        tree.GenerateFromMesh(essai, true);*/
        /*Material mat = new Material();
        mat.Ambient = Color.FromArgb(0, 0, 255);
        mat.Diffuse = Color.FromArgb(0, 0, 255);
        mat.Specular = Color.FromArgb(0, 0, 255);
        mat.Emissive = Color.FromArgb(0, 0, 255);
        KinemaMeshObject model = new KinemaMeshObject(moteur, new Vector3(50, 0
, 0), new Quaternion(1, 0, 0, 0), Vector3.Empty, new Vector3(0f, -1.0f, 1.0f), e
ssai, new Texture[] { null }, new Material[] { mat });
        model.GenerateOctree(5, 15);
        //model.GenerateBinaryTree();
        model.BoundingSphereTreeLevelDrawing = 5;
        model.Scaling = 10;
        moteur.GameObjects.Add(model);*/

        /*// Pendule double
        // Dock du pendule
        KinemaDockObject dock = new KinemaDockObject(moteur, new Vector3(150, 4
0, 0), Quaternion.Identity);
        // Premier élément du pendule Pendule
        Mesh penduleMesh = Mesh.Box(moteur.Device, 10, 10, 60);
        Material mat = new Material();
        mat.Ambient = Color.FromArgb(0, 0, 255);
        mat.Diffuse = Color.FromArgb(0, 0, 255);
        mat.Specular = Color.FromArgb(0, 0, 255);
        mat.Emissive = Color.FromArgb(0, 0, 255);
        KinemaMeshObject pendule1 = new KinemaMeshObject(moteur, new Vector3(15
0, 40, 30), Quaternion.Identity, Vector3.Empty, Vector3.Empty, penduleMesh, new
Texture[] { null }, new Material[] { mat });
        pendule1.Mass = 10.0f;
        pendule1.Forces = new Vector3(0, -100, 0);
        pendule1.SetBoxInertialTensor(0.10f, 0.10f, 0.60f);
        // Deuxième élément du pendule
        penduleMesh = Mesh.Box(moteur.Device, 10, 60, 10);
        KinemaMeshObject pendule2 = new KinemaMeshObject(moteur, new Vector3(15
0, 10, 60), Quaternion.Identity, Vector3.Empty, Vector3.Empty, penduleMesh, new
Texture[] { null }, new Material[] { mat });
        pendule2.Mass = 10.0f;
        pendule2.Forces = new Vector3(0, -100, 0);
        pendule2.SetBoxInertialTensor(0.10f, 0.60f, 0.10f);
        // Liaisons
        BallJoint joint1 = new BallJoint(dock, pendule1, new Vector3(0, 0, 0),
new Vector3(0, 0, -30), 0.0001f);
        BallJoint joint2 = new BallJoint(pendule1, pendule2, new Vector3(0, 0,
30), new Vector3(0, 30, 0), 0.0001f);
        // Assemblage
        AssemblyObject assembly = new AssemblyObject();
        assembly.AssemblyObjects.Add(dock);
        assembly.AssemblyObjects.Add(pendule1);
        assembly.AssemblyObjects.Add(pendule2);
        assembly.AssemblyJoints.Add(joint1);
        assembly.AssemblyJoints.Add(joint2);
        moteur.GameObjects.Add(assembly);*/

        // Essai BallSlider
        // Dock du pendule
        KinemaDockObject dock = new KinemaDockObject(moteur, new Vector3(150, 4
0, 0), Quaternion.Identity);
        // Premier élément du pendule Pendule
        Mesh penduleMesh = Mesh.Box(moteur.Device, 10, 10, 60);
        Material mat = new Material();
```

```
         mat.Ambient = Color.FromArgb(0, 0, 255);
         mat.Diffuse = Color.FromArgb(0, 0, 255);
         mat.Specular = Color.FromArgb(0, 0, 255);
         mat.Emissive = Color.FromArgb(0, 0, 255);
         KinemaMeshObject pendule1 = new KinemaMeshObject(moteur, new Vector3(15
0, 40, 30), Quaternion.Identity, Vector3.Empty, Vector3.Empty, penduleMesh, new
Texture[] { null }, new Material[] { mat });
         pendule1.Mass = 10.0f;
         //pendule1.Forces = new Vector3(0, -100, 0);
         pendule1.SetBoxInertialTensor(0.10f, 0.10f, 0.60f);
         // Objet coulissant
         penduleMesh = Mesh.Box(moteur.Device, 20, 20, 20);
         KinemaMeshObject pendule2 = new KinemaMeshObject(moteur, new Vector3(15
0, 40, 40), Quaternion.Identity, Vector3.Empty, Vector3.Empty, penduleMesh, new
Texture[] { null }, new Material[] { mat });
         pendule2.Mass = 5.0f;
         pendule2.Forces = new Vector3(0, -50, 0);
         pendule2.SetBoxInertialTensor(0.20f, 0.20f, 0.20f);
         // Liaisons
         BallJoint joint1 = new BallJoint(dock, pendule1, new Vector3(0, 0, 0),
new Vector3(0, 0, -30), 0.0001f);
         BallSlider joint2 = new BallSlider(pendule1, pendule2, new Vector3(0, 0
, 0), new Vector3(0, 0, 0), new Vector3(0, 0, 1), 1f);
         // Assemblage
         AssemblyObject assembly = new AssemblyObject();
         assembly.AssemblyObjects.Add(dock);
         assembly.AssemblyObjects.Add(pendule1);
         assembly.AssemblyObjects.Add(pendule2);
         assembly.AssemblyJoints.Add(joint1);
         assembly.AssemblyJoints.Add(joint2);
         moteur.GameObjects.Add(assembly);
      }

      public void UpdateEngine()
      {
         moteur.UpdateEngine();
      }

      private void MainForm_DoubleClick(object sender, EventArgs e)
      {
         moteur.IsPaused = !moteur.IsPaused;
      }

      protected override void OnPaint(PaintEventArgs e)
      {
         // On met à jour le moteur
         moteur.UpdateEngine();

         // On affiche le fps dans la barre de titre
         //this.Text = string.Format("{0} fps", moteur.FrameRate());

         this.Invalidate();
      }
   }
}

/*public void OnApplicationIdle(object sender, EventArgs e)
      {
         while (AppStillIdle)
         {
            // Render a frame during idle time (no messages are waiting)
            moteur.UpdateEngine();
         }
      }

private bool AppStillIdle
```

```
{
   get
   {
      NativeMethods.Message msg;
      return !NativeMethods.PeekMessage(out msg, IntPtr.Zero, 0, 0, 0);
   }
}

public class NativeMethods
{
   #region Win32 User Messages / Structures

   /// <summary>Window messages</summary>
   public enum WindowMessage : uint
   {
      // Misc messages
      Destroy = 0x0002,
      Close = 0x0010,
      Quit = 0x0012,
      Paint = 0x000F,
      SetCursor = 0x0020,
      ActivateApplication = 0x001C,
      EnterMenuLoop = 0x0211,
      ExitMenuLoop = 0x0212,
      NonClientHitTest = 0x0084,
      PowerBroadcast = 0x0218,
      SystemCommand = 0x0112,
      GetMinMax = 0x0024,

      // Keyboard messages
      KeyDown = 0x0100,
      KeyUp = 0x0101,
      Character = 0x0102,
      SystemKeyDown = 0x0104,
      SystemKeyUp = 0x0105,
      SystemCharacter = 0x0106,

      // Mouse messages
      MouseMove = 0x0200,
      LeftButtonDown = 0x0201,
      LeftButtonUp = 0x0202,
      LeftButtonDoubleClick = 0x0203,
      RightButtonDown = 0x0204,
      RightButtonUp = 0x0205,
      RightButtonDoubleClick = 0x0206,
      MiddleButtonDown = 0x0207,
      MiddleButtonUp = 0x0208,
      MiddleButtonDoubleClick = 0x0209,
      MouseWheel = 0x020a,
      XButtonDown = 0x020B,
      XButtonUp = 0x020c,
      XButtonDoubleClick = 0x020d,
      MouseFirst = LeftButtonDown, // Skip mouse move, it happens a lot and ther
e is another message for that
      MouseLast = XButtonDoubleClick,

      // Sizing
      EnterSizeMove = 0x0231,
      ExitSizeMove = 0x0232,
      Size = 0x0005,

   }

   /// <summary>Mouse buttons</summary>
   public enum MouseButtons
   {
```

```
      Left = 0x0001,
      Right = 0x0002,
      Middle = 0x0010,
      Side1 = 0x0020,
      Side2 = 0x0040,
   }

   /// <summary>Windows Message</summary>
   [StructLayout(LayoutKind.Sequential)]
   public struct Message
   {
      public IntPtr hWnd;
      public WindowMessage msg;
      public IntPtr wParam;
      public IntPtr lParam;
      public uint time;
      public System.Drawing.Point p;
   }

   /// <summary>MinMax Info structure</summary>
   [StructLayout(LayoutKind.Sequential)]
   public struct MinMaxInformation
   {
      public System.Drawing.Point reserved;
      public System.Drawing.Point MaxSize;
      public System.Drawing.Point MaxPosition;
      public System.Drawing.Point MinTrackSize;
      public System.Drawing.Point MaxTrackSize;
   }

   /// <summary>Monitor Info structure</summary>
   [StructLayout(LayoutKind.Sequential)]
   public struct MonitorInformation
   {
      public uint Size; // Size of this structure
      public System.Drawing.Rectangle MonitorRectangle;
      public System.Drawing.Rectangle WorkRectangle;
      public uint Flags; // Possible flags
   }
   #endregion

   #region Windows API calls
   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [System.Runtime.InteropServices.DllImport("winmm.dll")]
   public static extern IntPtr timeBeginPeriod(uint period);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("kernel32")]
   public static extern bool QueryPerformanceFrequency(ref long PerformanceFrequ
ency);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("kernel32")]
   public static extern bool QueryPerformanceCounter(ref long PerformanceCount);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern bool GetMonitorInfo(IntPtr hWnd, ref MonitorInformation
info);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
```

```
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern IntPtr MonitorFromWindow(IntPtr hWnd, uint flags);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern short GetAsyncKeyState(uint key);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern IntPtr SetCapture(IntPtr handle);

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern bool ReleaseCapture();

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern int GetCaretBlinkTime();

   [System.Security.SuppressUnmanagedCodeSecurity] // We won't use this maliciou
sly
   [DllImport("User32.dll", CharSet = CharSet.Auto)]
   public static extern bool PeekMessage(out Message msg, IntPtr hWnd, uint mess
ageFilterMin, uint messageFilterMax, uint flags);
   #endregion

   #region Class Methods
   private NativeMethods() { } // No creation
   /// <summary>Returns the low word</summary>
   public static short LoWord(uint l)
   {
      return (short)(l & 0xffff);
   }
   /// <summary>Returns the high word</summary>
   public static short HiWord(uint l)
   {
      return (short)(l >> 16);
   }

   /// <summary>Makes two shorts into a long</summary>
   public static uint MakeUInt32(short l, short r)
   {
      return (uint)((l & 0xffff) | ((r & 0xffff) << 16));
   }

   /// <summary>Is this key down right now</summary>
   public static bool IsKeyDown(System.Windows.Forms.Keys key)
   {
      return (GetAsyncKeyState((int)System.Windows.Forms.Keys.ShiftKey) & 0x8000
) != 0;
   }
   #endregion
}*/
```

```
namespace KinemaExample
{
   partial class MainForm
   {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed;
otherwise, false.</param>
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
         base.Dispose(disposing);
      }

      #region Windows Form Designer generated code

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.SuspendLayout();
         //
         // MainForm
         //
         this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Inherit;
         this.ClientSize = new System.Drawing.Size(244, 243);
         this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle
;
         this.MaximizeBox = false;
         this.Name = "MainForm";
         this.Text = "Form1";
         this.DoubleClick += new System.EventHandler(this.MainForm_DoubleClick);
         this.ResumeLayout(false);

      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using System.Windows.Forms;
using System.Drawing;

namespace Kinema
{
   /// <summary>
   /// Classe représentant un moteur physique et graphique
   /// </summary>
   public class KinemaEngine
   {
      #region Variables de la classe

      // Variables systeme
      protected Form fenetrePrinc = null;
      protected Device device;
      protected Microsoft.DirectX.Direct3D.Font statsFont;

      // Variables d'état
      protected double deltaTime;
      protected int frameRate;
      protected bool isDrawingStats = false, isRendingCollisions = false, isPaus
ed = false;

      // Eléments du moteur
      protected Camera camera;
      protected Mouse mouse;
      protected Keyboard keyboard;
      protected List<KinemaBaseObject> gameObjects = new List<KinemaBaseObject>(
);

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur
      /// </summary>
      /// <param name="fenetre">La fenêtre hôte de l'application</param>
      public KinemaEngine(Form fenetre)
      {
         FenetrePrinc = fenetre;

         // Création du device
         ConfigureDevice();

         // Création des devices de la souris et du clavier
         Mouse = new Mouse(FenetrePrinc, false);
         Keyboard = new Keyboard(FenetrePrinc);

         HiResTimer.Reset();
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Affiche les statistiques du moteur si mise à <see langword="true"/>
      /// </summary>
      public bool IsDrawingStats
      {
```

```
         get
         {
            return isDrawingStats;
         }
         set
         {
            isDrawingStats = value;
         }
      }

      /// <summary>
      /// Affiche les spheres des arbres en contact si mise à <see langword="tru
e"/>
      /// </summary>
      public bool IsRenderingCollisions
      {
         get
         {
            return isRendingCollisions;
         }
         set
         {
            isRendingCollisions = value;
         }
      }

      /// <summary>
      /// Permet de mettre en pause le moteur physique
      /// </summary>
      public bool IsPaused
      {
         get
         {
            return isPaused;
         }
         set
         {
            isPaused = value;
            if (!isPaused)
            {
               HiResTimer.Reset();
            }
         }
      }

      /// <summary>
      /// Fenêtre hôte de l'application
      /// </summary>
      public Form FenetrePrinc
      {
         get
         {
            return fenetrePrinc;
         }
         protected set
         {
            fenetrePrinc = value;
         }
      }

      /// <summary>
      /// Device DirectX utilisé pour le rendu
      /// </summary>
      public Device Device
      {
         get
```

```
        {
            return device;
        }
    }

    /// <summary>
    /// Camera utilisée pour le rendu graphique
    /// </summary>
    public Camera Camera
    {
        get
        {
            return camera;
        }
        set
        {
            camera = value;
        }
    }

    /// <summary>
    /// Souris utilisée pour les entrées utilisateur
    /// </summary>
    public Mouse Mouse
    {
        get
        {
            return mouse;
        }
        protected set
        {
            mouse = value;
        }
    }

    /// <summary>
    /// Clavier utilisé pour les entrées utilisateur
    /// </summary>
    public Keyboard Keyboard
    {
        get
        {
            return keyboard;
        }
        protected set
        {
            keyboard = value;
        }
    }

    /// <summary>
    /// Liste de tous les objets gérés par le moteur
    /// </summary>
    public List<KinemaBaseObject> GameObjects
    {
        get
        {
            return gameObjects;
        }
    }

    #endregion

    #region Gestion du device et des ressources

    /// <summary>
```

```
      /// Configure le device en vue de son utilisation pour le rendu
      /// </summary>
      protected virtual void ConfigureDevice()
      {
         // On récupère le numéro de l'adapter par défaut
         int adapterOrdinal = Manager.Adapters.Default.Adapter;

         // On récupère les "capabilities" du device pour les établir nos "Creat
eFlags"
         Caps caps = Microsoft.DirectX.Direct3D.Manager.GetDeviceCaps(adapterOrd
inal, Microsoft.DirectX.Direct3D.DeviceType.Hardware);
         CreateFlags createFlags;

         // On vérifie si la carte autorise le traitement matériel
         if (caps.DeviceCaps.SupportsHardwareTransformAndLight)
         {
            createFlags = CreateFlags.HardwareVertexProcessing;
         }
         else
         {
            createFlags = CreateFlags.SoftwareVertexProcessing;
         }

         // On vérifie si la carte autorise les opérations de rasterization, tra
nsformations matricielles, lumières et ombres
         if (caps.DeviceCaps.SupportsPureDevice && createFlags == CreateFlags.Ha
rdwareVertexProcessing)
         {
            createFlags |= CreateFlags.PureDevice;
         }

         // On établit les "PresentParameters" qui déterminent le comportement d
u device
         PresentParameters presentParams = new PresentParameters();
         presentParams.SwapEffect = SwapEffect.Discard;
         presentParams.EnableAutoDepthStencil = true;
         presentParams.AutoDepthStencilFormat = DepthFormat.D16;
         presentParams.PresentationInterval = PresentInterval.Immediate;

         // On se place en mode fenêtre si l'on est en mode de déboggage
         presentParams.Windowed = true;
//#if DEBUG
//         presentParams.Windowed = true;
//#else
//            presentParams.Windowed = false;
//#endif

         // On crée le device
         device = new Device(adapterOrdinal, DeviceType.Hardware, fenetrePrinc,
            createFlags | CreateFlags.FpuPreserve, presentParams);

         device.DeviceReset += new EventHandler(OnDeviceReset);
         device.DeviceLost += new EventHandler(OnDeviceLost);
         device.Disposing += new EventHandler(OnDeviceDisposing);

         // On crée la font qui permet d'afficher les statistiques
         statsFont = new Microsoft.DirectX.Direct3D.Font(device, 15, 0, FontWeig
ht.Bold,
               1, false, CharacterSet.Default, Precision.Default, FontQuality.D
efault,
               PitchAndFamily.DefaultPitch | PitchAndFamily.FamilyDoNotCare, "A
rial");

         OnDeviceReset(fenetrePrinc, null);
      }
```

```
        /// <summary>
        /// Gère le reset du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        protected virtual void OnDeviceReset(object sender, EventArgs e)
        {
            // On active le z-buffer.
            device.RenderState.ZBufferEnable = true;

            // On active l'alpha blending
            device.RenderState.AlphaBlendEnable = true;
            device.RenderState.SourceBlend = Blend.SourceColor;
            device.RenderState.DestinationBlend = Blend.InvSourceAlpha;

            // On active la gestion des lumières
            device.RenderState.Lighting = true;

            // On active une lumière ambiante blanche
            device.RenderState.Ambient = System.Drawing.Color.White;

            // On désactive le culling pour pouvoir voir toutes les faces des objet
s
            device.RenderState.CullMode = Cull.None;

            // On étend l'évènement à tous les objets
            if (statsFont != null) statsFont.OnResetDevice();
            foreach (KinemaBaseObject gameObject in GameObjects)
            {
                gameObject.OnDeviceReset(sender, e);
            }

        }

        /// <summary>
        /// Gère la perte du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        protected virtual void OnDeviceLost(object sender, EventArgs e)
        {
            // On étend l'évènement à tous les objets
            if (statsFont != null && !statsFont.Disposed) statsFont.OnLostDevice();
            foreach (KinemaBaseObject gameObject in GameObjects)
            {
                gameObject.OnDeviceLost(sender, e);
            }
        }

        /// <summary>
        /// Gère la libération des ressources à la destruction du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        protected virtual void OnDeviceDisposing(object sender, EventArgs e)
        {
            // On étend l'évènement à tous les objets
            if (statsFont != null) statsFont.Dispose();
            foreach (KinemaBaseObject gameObject in GameObjects)
            {
                gameObject.OnDeviceDisposing(sender, e);
            }

            Mouse.Dispose();
            Keyboard.Dispose();
        }
```

```
      #endregion

      #region Opérations de mise à jour

      /// <summary>
      /// Met à jour le moteur et lance le rendu graphique
      /// </summary>
      public virtual void UpdateEngine()
      {
         // Si le moteur n'est pas en pause, on gère le chrono, sinon deltatime
est nul
         if (IsPaused)
         {
            deltaTime = 0;
         }
         else
         {
            deltaTime = HiResTimer.GetElapsedTime();
            HiResTimer.Start();
         }

         // On calcule le framerate
         frameRate = FrameRateHelper.CalculateFrameRate();

         // On gère les entrées utilisateur
         UserInputs(deltaTime);

         // Mise à jour des objets si l'on est pas en pause
         if (!IsPaused)
            UpdateObjects(deltaTime);

         // On efface l'écran
         device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.Black, 1.0f,
 0);

         // On met à jour la caméra et on définit les matrices de vue et project
ion
         camera.Update(deltaTime);
         device.Transform.View = camera.ViewMatrix;
         device.Transform.Projection = camera.ProjectionMatrix;

         // On commence le dessin
         device.BeginScene();

         // On dessine
         Render();

         // On affiche les statistiques si demandé
         if (IsDrawingStats)
         {
            statsFont.DrawText(null, frameRate + " fps", new System.Drawing.Rect
angle(10, 10, 0, 0),
                  DrawTextFormat.NoClip, System.Drawing.Color.Yellow);
         }

         // On affiche l'état du moteur (pause ou non)
         if (IsPaused)
         {
            statsFont.DrawText(null, "PAUSE", new System.Drawing.Rectangle(fenet
rePrinc.Size.Width / 2, fenetrePrinc.Size.Height / 2, 0, 0),
                  DrawTextFormat.NoClip | DrawTextFormat.Center, System.Drawin
g.Color.Red);
         }

         // On termine le dessin
```

```
         device.EndScene();

         // On affiche la scène à l'écran
          device.Present();
       }

      /// <summary>
      /// Gère les entrées utilisateur
      /// </summary>
      /// <param name="updateTime">Durée du pas de calcul de la frame</param>
      protected virtual void UserInputs(double updateTime)
      {
         // On récupère les entrées utilisateur
         Mouse.Poll(updateTime);
         Keyboard.Poll();
         // On quitte si la touche escape est appuyée
         if (Keyboard.State[Microsoft.DirectX.DirectInput.Key.Escape])
         {
            Application.Exit();
         }
      }

      /// <summary>
      /// Met à jour les objets gérés par le moteur
      /// </summary>
      /// <param name="updateTime">Durée du pas de calcul de la frame</param>
      protected virtual void UpdateObjects(double updateTime)
      {
         // On gère les collisions
         // Pour chaque objet d'index i de [0, nbrobjets − 1] on teste les colli
sions avec les objets d'index j>i
         Vector3 normal;
         for (int i = 0; i < GameObjects.Count; i++)
            for (int j = i + 1; j < GameObjects.Count; j++)
            {
               // On teste la collision
               if (KinemaBaseObject.IsColliding(GameObjects[i], GameObjects[j],
out normal))
               {
                  // TODO : gérer la collision :p
               }
            }

         // On met à jour tous les objets gérés par le moteur
         foreach (KinemaBaseObject gameObject in GameObjects)
         {
            gameObject.Update(updateTime);
            while (!gameObject.IsValid)
               gameObject.Update(updateTime);
            gameObject.FinalizeUpdate();
         }
      }

      /// <summary>
      /// Dessine la scène à l'écran
      /// </summary>
      protected virtual void Render()
      {
         // On affiche tous les objets gérés par le moteur
         foreach (KinemaBaseObject gameObject in GameObjects)
         {
            gameObject.Render();
         }
      }

      #endregion
```

```
      #region Méthodes d’accès aux statistiques

      /// <summary>
      /// Renvoie le nombre de fps du moteur.
      /// Cette valeur est recalculée à chaque appel à la fonction update()
      /// </summary>
      /// <returns></returns>
      public int FrameRate()
      {
         return frameRate;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Gère une caméra basique pour une scène 3D
   /// </summary>
   public class Camera
   {
      #region Variables de la classe

      protected float fov, nearPlane, farPlane;
      protected Vector3 position = new Vector3(), lookDirection = new Vector3(),
 cameraUp = new Vector3();
      protected KinemaEngine engine = null;
      protected Form fenetre = null;
      protected Matrix viewMatrix, projectionMatrix;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur avec paramètres par défaut de la caméra
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      public Camera(KinemaEngine kinemaEngine, Form fenetreRendu, Vector3 camera
Position, Vector3 cameraLookDir, Vector3 cameraUpVector)
         : this(kinemaEngine, fenetreRendu, cameraPosition, cameraLookDir, camer
aUpVector, (float)Math.PI / 4.0f, 1.0f, 100.0f) { }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      /// <param name="fieldOfVision">Angle du champ de vision de la caméra, en
radians</param>
      /// <param name="nearPlaneCoord">Distance de la caméra au plan limite proc
he du rendu</param>
      /// <param name="farPlaneCoord">Distance de la caméra au plan limite éloig
né du rendu</param>
      public Camera(KinemaEngine kinemaEngine, Form fenetreRendu, Vector3 camera
Position, Vector3 cameraLookDir, Vector3 cameraUpVector,
         float fieldOfVision, float nearPlaneCoord, float farPlaneCoord)
      {
         engine = kinemaEngine;
         fenetre = fenetreRendu;
         FieldOfVision = fieldOfVision;
         nearPlane = nearPlaneCoord;
         farPlane = farPlaneCoord;
         Position = cameraPosition;
```

```
            LookDirection = cameraLookDir;
            CameraUp = cameraUpVector;

            UpdateMatrices();
        }

        #endregion

        #region Propriétés et accesseurs

        /// <summary>
        /// Angle du champ de vision de la caméra, en radians
        /// </summary>
        public float FieldOfVision
        {
            get
            {
                return fov;
            }
            set
            {
                fov = NormalizeAngle(value);
            }
        }

        /// <summary>
        /// Distance de la caméra au plan limite proche du rendu
        /// </summary>
        public float NearPlane
        {
            get
            {
                return nearPlane;
            }
            set
            {
                nearPlane = value;
            }
        }

        /// <summary>
        /// Distance de la caméra au plan limite éloigné du rendu
        /// </summary>
        public float FarPlane
        {
            get
            {
                return farPlane;
            }
            set
            {
                farPlane = value;
            }
        }

        /// <summary>
        /// Ratio de la fenêtre de rendu
        /// </summary>
        public float AspectRatio
        {
            get
            {
                return (float)fenetre.Width / (float)fenetre.Height;
            }
        }
```

```
        /// <summary>
        /// Coordonnée de la position de la caméra selon l'axe X
        /// </summary>
        public float X
        {
            get
            {
                return position.X;
            }
            set
            {
                position.X = value;
            }
        }

        /// <summary>
        /// Coordonnée de la position de la caméra selon l'axe Y
        /// </summary>
        public float Y
        {
            get
            {
                return position.Y;
            }
            set
            {
                position.Y = value;
            }
        }

        /// <summary>
        /// Coordonnée de la position de la caméra selon l'axe Z
        /// </summary>
        public float Z
        {
            get
            {
                return position.Z;
            }
            set
            {
                position.Z = value;
            }
        }

        /// <summary>
        /// Position de la caméra
        /// </summary>
        public Vector3 Position
        {
            get
            {
                return new Vector3(X, Y, Z);
            }
            set
            {
                position.X = value.X;
                position.Y = value.Y;
                position.Z = value.Z;
            }
        }

        /// <summary>
        /// Direction dans laquelle regarde la caméra
        /// </summary>
        public Vector3 LookDirection
```

```
      {
         get
         {
            return new Vector3(lookDirection.X, lookDirection.Y, lookDirection.Z
);
         }
         set
         {
            lookDirection.X = value.X;
            lookDirection.Y = value.Y;
            lookDirection.Z = value.Z;
         }
      }

      /// <summary>
      /// Direction du haut de la caméra
      /// </summary>
      public Vector3 CameraUp
      {
         get
         {
            return new Vector3(cameraUp.X, cameraUp.Y, cameraUp.Z);
         }
         set
         {
            cameraUp.X = value.X;
            cameraUp.Y = value.Y;
            cameraUp.Z = value.Z;
         }
      }

      /// <summary>
      /// Matrice de projection de la caméra
      /// </summary>
      public Matrix ProjectionMatrix
      {
         get
         {
            return projectionMatrix;
         }
      }

      /// <summary>
      /// Matrice de vue de la caméra
      /// </summary>
      public Matrix ViewMatrix
      {
         get
         {
            return viewMatrix;
         }
      }

      #endregion

      #region Opérations de mise à jour

      /// <summary>
      /// Met à jour la caméra
      /// </summary>
      /// <param name="deltatime">Durée du pas de calcul de la frame</param>
      public virtual void Update(double deltatime)
      {
         // Cette classe de base ne fait pas évoluer la caméra, on se contente d
e mettre à jour les matrices
         UpdateMatrices();
```

```
      }

      /// <summary>
      /// Met à jour les matrices de projection de la caméra
      /// </summary>
      protected virtual void UpdateMatrices()
      {
         // Calcul de la matrice de projection
         projectionMatrix = Matrix.PerspectiveFovLH(FieldOfVision, AspectRatio,
NearPlane, FarPlane);

         // Calcul de la matrice de vue
         viewMatrix = Matrix.LookAtLH(position, position + lookDirection, camera
Up);
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Normalise un angle
      /// </summary>
      /// <param name="angle">Angle à normaliser</param>
      /// <returns>Vaeur de l'angle ramenée dans l'intervalle [0,2.Pi[</returns>
      protected float NormalizeAngle(float angle)
      {
         while (angle >= 2.0f * (float)Math.PI)
            angle -= 2.0f * (float)Math.PI;

         while (angle < 0.0f)
            angle += 2.0f * (float)Math.PI;

         return angle;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Caméra possédant des méthodes de déplacements simpels
   /// yaw   -> rotation autour de Y (cameraUp)
   /// pitch -> rotation autour de X (right = cameraUp^lookDir)
   /// roll  -> rotation autour de Z (lookDir)
   /// </summary>
   public class MovableCamera : Camera
   {
      #region Constructeur(s)

      // Constructeur avec paramètres de la caméra par défaut
      /// <summary>
      /// Constructeur avec paramètres de la caméra par défaut
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      public MovableCamera(KinemaEngine kinemaEngine, Form fenetreRendu, Vector3
 cameraPosition, Vector3 cameraLookDir, Vector3 cameraUpVector)
         : base(kinemaEngine, fenetreRendu, cameraPosition, cameraLookDir, camer
aUpVector, (float)Math.PI / 4.0f, 1.0f, 100.0f) { }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      /// <param name="fieldOfVision">Angle du champ de vision de la caméra, en
radians</param>
      /// <param name="nearPlaneCoord">Distance de la caméra au plan limite proc
he du rendu</param>
      /// <param name="farPlaneCoord">Distance de la caméra au plan limite éloig
né du rendu</param>
      public MovableCamera(KinemaEngine kinemaEngine, Form fenetreRendu, Vector3
 cameraPosition, Vector3 cameraLookDir, Vector3 cameraUpVector,
         float fieldOfVision, float nearPlaneCoord, float farPlaneCoord)
         : base(kinemaEngine, fenetreRendu, cameraPosition, cameraLookDir, camer
aUpVector, fieldOfVision, nearPlaneCoord, farPlaneCoord) { }

      #endregion

      #region Méthodes de déplacement

      /// <summary>
      /// Déplace la caméra vers l'avant
      /// </summary>
      /// <param name="unit">Valeur algébrique du déplacement</param>
      public void MoveForward(float unit)
      {
         Vector3 deplacement = Vector3.Normalize(lookDirection);
```

```
         position = position + unit * deplacement;
      }

      /// <summary>
      /// Déplace la caméra vers le côté droit
      /// </summary>
      /// <param name="unit">Valeur algébrique du déplacement</param>
      public void Strafe(float unit)
      {
         Vector3 deplacement = Vector3.Cross(cameraUp, lookDirection);
         deplacement = Vector3.Normalize(deplacement);

         position = position + unit * deplacement;
      }

      /// <summary>
      /// Déplace la caméra vers le haut
      /// </summary>
      /// <param name="unit">Valeur algébrique du déplacement</param>
      public void MoveUp(float unit)
      {
         Vector3 deplacement = Vector3.Normalize(cameraUp);

         position = position + unit * deplacement;
      }

      /// <summary>
      /// Modifie le Yaw
      /// </summary>
      /// <param name="angle">Angle à ajouter</param>
      public void AdjustYaw(float angle)
      {
         if (angle == 0.0f) return;

         Matrix rotation = Matrix.RotationAxis(cameraUp, angle);

         lookDirection.TransformNormal(rotation);
      }

      /// <summary>
      /// Modifie le Pitch
      /// </summary>
      /// <param name="angle">Angle à ajouter</param>
      public void AdjustPitch(float angle)
      {
         if (angle == 0.0f) return;

         Matrix rotation = Matrix.RotationAxis(Vector3.Cross(cameraUp, lookDirec
tion), angle);

         lookDirection.TransformNormal(rotation);
         cameraUp.TransformNormal(rotation);
      }

      /// <summary>
      /// Modifie le Roll
      /// </summary>
      /// <param name="angle">Angle à ajouter</param>
      public void AdjustRoll(float angle)
      {
         if (angle == 0.0f) return;

         Matrix rotation = Matrix.RotationAxis(lookDirection, angle);

         cameraUp.TransformNormal(rotation);
```

```
        }

        #endregion
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using Microsoft.DirectX.DirectInput;

namespace Kinema
{
   /// <summary>
   /// Caméra gérée par l'utilisateur avec le clavier et la souris
   /// Comporte une méthode de mapping des touches
   /// </summary>
   public class UserControledCamera : MovableCamera
   {
      #region Variables de la classe

      protected Key[] keyMapping = new Key[6];
      protected MoveLookAtCondition lookAtCondition;
      protected float[] mouvingSpeed = new float[] { 20.0f, -20.0f, 20.0f, -20.0
f, 20.0f, -20.0f };
      protected float[] rotatingScaling = new float[] { 0.01f, 0.01f, 0.01f };

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur avec paramètres par défaut de la caméra
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      public UserControledCamera(KinemaEngine kinemaEngine, Form fenetreRendu, V
ector3 cameraPosition, Vector3 cameraLookDir, Vector3 cameraUpVector)
         : base(kinemaEngine, fenetreRendu, cameraPosition, cameraLookDir, camer
aUpVector, (float)Math.PI / 4.0f, 1.0f, 100.0f)
      {
         lookAtCondition = MoveLookAtCondition.None;
         SetDefaultKeyMapping(DefaultMapping.None);
      }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel appartient la caméra</param>
      /// <param name="fenetreRendu">Fenêtre hôte de l'application</param>
      /// <param name="cameraPosition">Position de la caméra</param>
      /// <param name="cameraLookDir">Direction dans laquelle la caméra regarde<
/param>
      /// <param name="cameraUpVector">Direction du haut de la caméra</param>
      /// <param name="fieldOfVision">Angle du champ de vision de la caméra, en
radians</param>
      /// <param name="nearPlaneCoord">Distance de la caméra au plan limite proc
he du rendu</param>
      /// <param name="farPlaneCoord">Distance de la caméra au plan limite éloig
né du rendu</param>
      public UserControledCamera(KinemaEngine kinemaEngine, Form fenetreRendu, V
ector3 cameraPosition, Vector3 cameraLookDir, Vector3 cameraUpVector,
         float fieldOfVision, float nearPlaneCoord, float farPlaneCoord)
         : base(kinemaEngine, fenetreRendu, cameraPosition, cameraLookDir, camer
aUpVector, fieldOfVision, nearPlaneCoord, farPlaneCoord)
```

```
        {
            lookAtCondition = MoveLookAtCondition.None;
            SetDefaultKeyMapping(DefaultMapping.None);
        }

        #endregion

        #region Propriétés et accesseurs

        /// <summary>
        /// Tableau contenant les codes des touches de mouvement
        /// </summary>
        public Key[] KeyMapping
        {
            get
            {
                return keyMapping;
            }
        }

        /// <summary>
        /// Vitesses de déplacements dans les 3 directions
        /// </summary>
        public float[] MouvingSpeed
        {
            get
            {
                return mouvingSpeed;
            }
            set
            {
                mouvingSpeed = value;
            }
        }

        /// <summary>
        /// Valeurs de sensibilités de la souris pour les 3 rotations
        /// </summary>
        public float[] RotatingScaling
        {
            get
            {
                return rotatingScaling;
            }
            set
            {
                rotatingScaling = value;
            }
        }

        /// <summary>
        /// Condition d'activation de la commande à la souris
        /// </summary>
        public MoveLookAtCondition LookAtCondition
        {
            get
            {
                return lookAtCondition;
            }
            set
            {
                lookAtCondition = value;
            }
        }

        #endregion
```

```
      #region Opérations de mise à jour

      /// <summary>
      /// Met à jour la caméra
      /// </summary>
      /// <param name="deltatime">Durée du pas de calcul de la frame</param>
      public override void Update(double deltatime)
      {
         float dt = (float)deltatime;

         // On gère les entrées utilisateur
         // entrées clavier
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.Forward]])
         {
            MoveForward(dt * mouvingSpeed[(int)MouvementKeyMapping.Forward]);
         }
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.Backward]])
         {
            MoveForward(dt * mouvingSpeed[(int)MouvementKeyMapping.Backward]);
         }
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.StrafeRight]])
         {
            Strafe(dt * mouvingSpeed[(int)MouvementKeyMapping.StrafeRight]);
         }
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.StrafeLeft]])
         {
            Strafe(dt * mouvingSpeed[(int)MouvementKeyMapping.StrafeLeft]);
         }
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.Up]])
         {
            MoveUp(dt * mouvingSpeed[(int)MouvementKeyMapping.Up]);
         }
         if (engine.Keyboard[KeyMapping[(int)MouvementKeyMapping.Down]])
         {
            MoveUp(dt * mouvingSpeed[(int)MouvementKeyMapping.Down]);
         }

         // entrées souris
         if (IsLookAtConditionVerified())
         {
            AdjustYaw(engine.Mouse.Dx * rotatingScaling[(int)Rotation.Yaw]);
            AdjustPitch(engine.Mouse.Dy * rotatingScaling[(int)Rotation.Pitch]);
            AdjustRoll(engine.Mouse.Dz * rotatingScaling[(int)Rotation.Roll]);
         }

         // On appelle la méthode de la classe mère qui met à jour les matrices
de vue et projection
         base.Update(deltatime);
      }

      #endregion

      #region méthodes helper

      // renvoie true si le mapping est reconnu, false sinon
      /// <summary>
      /// Permet d'appliquer un mapping par défaut
      /// </summary>
      /// <param name="mapping">Mapping des touches à appliquer</param>
      /// <returns><see langword="true"/> si le mapping est reconnu, <see langwo
rd="false"/> sinon</returns>
      public virtual bool SetDefaultKeyMapping(DefaultMapping mapping)
      {
         switch (mapping)
         {
```

```
            case DefaultMapping.None:
               KeyMapping[(int)MouvementKeyMapping.Forward] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Backward] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.StrafeRight] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.StrafeLeft] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Up] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Down] = (Key)(-1);
               return true;
            case DefaultMapping.WASD:
               KeyMapping[(int)MouvementKeyMapping.Forward] = Key.W;
               KeyMapping[(int)MouvementKeyMapping.Backward] = Key.S;
               KeyMapping[(int)MouvementKeyMapping.StrafeRight] = Key.D;
               KeyMapping[(int)MouvementKeyMapping.StrafeLeft] = Key.A;
               KeyMapping[(int)MouvementKeyMapping.Up] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Down] = (Key)(-1);
               return true;
            case DefaultMapping.ZQSD:
               KeyMapping[(int)MouvementKeyMapping.Forward] = Key.Z;
               KeyMapping[(int)MouvementKeyMapping.Backward] = Key.S;
               KeyMapping[(int)MouvementKeyMapping.StrafeRight] = Key.D;
               KeyMapping[(int)MouvementKeyMapping.StrafeLeft] = Key.Q;
               KeyMapping[(int)MouvementKeyMapping.Up] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Down] = (Key)(-1);
               return true;
            case DefaultMapping.Arrows:
               KeyMapping[(int)MouvementKeyMapping.Forward] = Key.UpArrow;
               KeyMapping[(int)MouvementKeyMapping.Backward] = Key.DownArrow;
               KeyMapping[(int)MouvementKeyMapping.StrafeRight] = Key.RightArrow
;
               KeyMapping[(int)MouvementKeyMapping.StrafeLeft] = Key.LeftArrow;
               KeyMapping[(int)MouvementKeyMapping.Up] = (Key)(-1);
               KeyMapping[(int)MouvementKeyMapping.Down] = (Key)(-1);
               return true;
            default:
               return false;
         }
      }

      /// <summary>
      /// Détermine si la condition d'activation de la commande à la souris est
vérifiée
      /// </summary>
      /// <returns><see langword="true"/> si la condition est vérifiée</returns>
      protected virtual bool IsLookAtConditionVerified()
      {
         if (LookAtCondition == MoveLookAtCondition.None)
            return true;
         if (LookAtCondition == MoveLookAtCondition.Never)
            return false;

         return (0 != engine.Mouse.MouseButtons[(int)LookAtCondition]);
      }

      #endregion

      #region Enumérations servant au système de mapping

      public enum MoveLookAtCondition { None = -2, Never = -1, Mouse1 = 0, Mouse
2 = 1, Mouse3 = 2}
      public enum MouvementKeyMapping { Forward = 0, Backward = 1, StrafeRight =
 2, StrafeLeft = 3, Up = 4, Down = 5 }
      public enum Rotation { Yaw = 0, Pitch = 1, Roll = 2 }
      public enum DefaultMapping { None = 0, WASD = 1, ZQSD = 2, Arrows = 3 }

      #endregion
   }
```

```
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe implémentant la résolution des problèmes de dynamique
   /// </summary>
   public class DynamicSolver
   {
      #region Variables de la classe

      protected Mapack.Matrix system = null, inversedSystem = null;
      protected AssemblyObject assemblyObject;

      #endregion

      #region Constructeur(s)

      public DynamicSolver(AssemblyObject assembly)
      {
         assemblyObject = assembly;
      }

      #endregion

      #region Matrices d'application des impulsions/moments angulaires

      /// <summary>
      /// Calcule la matrice a* de produit vectoriel d'un vecteur a
      /// telle que a^b = a*.b
      /// </summary>
      /// <param name="vecteur">Vecteur dont il faut calculer la matrice de prod
uit vectoriel</param>
      /// <returns>Matrice de produit vectoriel du vecteur</returns>
      public static Mapack.Matrix CrossProductMatrix(Vector3 vecteur)
      {
         Mapack.Matrix matrix = new Mapack.Matrix(3, 3);
         matrix[1, 0] = vecteur.Z;
         matrix[2, 0] = -vecteur.Y;
         matrix[2, 1] = vecteur.X;

         matrix[0, 1] = -vecteur.Z;
         matrix[0, 2] = vecteur.Y;
         matrix[1, 2] = -vecteur.X;

         return matrix;
      }

      /// <summary>
      /// Calcule la matrice K(P,Q) dans l'état courant du solide, telle que si
l'on applique une impulsion p en Q,
      /// la variation de vitesse de P s'écrit K(P,Q).p
      /// </summary>
      /// <param name="solid">Solide dont il faut calculer la matrice</param>
      /// <param name="P">Point (en coordonnées monde) dont on veut connaître la
 nouvelle vitesse</param>
      /// <param name="Q">Point (en coordonnées monde) d'application de l'impuls
ion</param>
      /// <returns>Matrice K(P,Q) du solide relative aux points P et Q</returns>
      public static Mapack.Matrix KPQ(KinemaBaseObject solid, Vector3 P, Vector3
 Q)
      {
```

```
         if (!solid.IsDynamic)
            return new Mapack.Matrix(3, 3);

         Mapack.Matrix Jinverted = solid.GetInertialTensorInStatesWorldCoord().I
nverse;
         Mapack.Matrix rpstar = CrossProductMatrix(P − solid.State.Position);
         Mapack.Matrix rqstar = CrossProductMatrix(Q − solid.State.Position);

         Mapack.Matrix result = (new Mapack.Matrix(3, 3, 1)) * (1 / solid.Mass)
− rpstar * (Jinverted * rqstar);

         return result;
      }

      /// <summary>
      /// Calcule la matrice Li dans l'état dourant du solide, telle que si l'on
 applique un moment angulaire l au solide,
      /// la variation de vitesse angulaire s'écrit Li.l
      /// </summary>
      /// <param name="solid">Solide dont il faut calculer la matrice</param>
      /// <returns>Matrice Li du solide</returns>
      public static Mapack.Matrix Li(KinemaBaseObject solid)
      {
         if (!solid.IsDynamic)
            return new Mapack.Matrix(3, 3);

         return solid.GetInertialTensorInStatesWorldCoord().Inverse;
      }

      /// <summary>
      /// Calcule la matrice W(i,P) dans l'état courant du solide, telle que si
l'on applique une impulsion p en P,
      /// la variation de vitesse angulaire s'écrit W(i,P).p
      /// </summary>
      /// <param name="solid">Solide dont il faut calculer la matrice</param>
      /// <param name="P">Point (en coordonnées monde) d'application de l'impuls
ion</param>
      /// <returns>Matrice W(i,P) du solide</returns>
      public static Mapack.Matrix WiP(KinemaBaseObject solid, Vector3 P)
      {
         if (!solid.IsDynamic)
            return new Mapack.Matrix(3, 3);

         Mapack.Matrix Jinverted = solid.GetInertialTensorInStatesWorldCoord().I
nverse;
         Mapack.Matrix rpstar = CrossProductMatrix(P − solid.State.Position);

         return Jinverted * rpstar;
      }

      /// <summary>
      /// Calcule la matrice U(P,i) dans l'état courant du solide, telle que si
l'on applique un moment angulaire l au solide,
      /// la variation de vitesse de P s'écrit U(P,i).l
      /// </summary>
      /// <param name="solid">Solide dont il faut calculer la matrice</param>
      /// <param name="P">Point (en coordonnées monde) dont on veut connaître la
 nouvelle vitesse</param>
      /// <returns>Matrice U(P,i) du solide</returns>
      public static Mapack.Matrix UPi(KinemaBaseObject solid, Vector3 P)
      {
         if (!solid.IsDynamic)
            return new Mapack.Matrix(3, 3);

         Mapack.Matrix Jinverted = solid.GetInertialTensorInStatesWorldCoord().I
nverse;
```

```
         Mapack.Matrix rpstar = CrossProductMatrix(P − solid.State.Position);

         return −(rpstar * Jinverted);
      }

      #endregion

      #region Matrices de résolution des liaisons

      /// <summary>
      /// Calcule la matrice Nij(k, X, Y) relative à 2 liaisons ayant un solide
en commun
      /// </summary>
      /// <param name="joint1">Première liaison (i)</param>
      /// <param name="joint2">Seconde liaison (j)</param>
      /// <param name="commonSolid">Solide commun aux 2 liaisons (k)</param>
      /// <param name="joint1Point">Point de la première liaison (X)</param>
      /// <param name="joint2Point">Point de la seconde liaison (Y)</param>
      /// <returns>Matrice Nij(k, X, Y) relative aux 2 liaisons</returns>
      public static Mapack.Matrix Nij(BaseJoint joint1, BaseJoint joint2,
         KinemaBaseObject commonSolid, Vector3 joint1Point, Vector3 joint2Point)
      {
         if (joint1.Type == BaseJoint.JointType.TranslationalConstraint)
         {
            if (joint2.Type == BaseJoint.JointType.TranslationalConstraint)
               return KPQ(commonSolid, joint1Point, joint2Point);
            if (joint2.Type == BaseJoint.JointType.RotationalConstraint)
               return UPi(commonSolid, joint1Point);
         }
         if (joint1.Type == BaseJoint.JointType.RotationalConstraint)
         {
            if (joint2.Type == BaseJoint.JointType.RotationalConstraint)
               return Li(commonSolid);
            if (joint2.Type == BaseJoint.JointType.TranslationalConstraint)
               return WiP(commonSolid, joint2Point);
         }

         // Normalement, il est impossible d'arriver ici
         return new Mapack.Matrix(3, 3);
      }

      /// <summary>
      /// Calcule la matrice Mij relative à 2 liaisons qui traduit les inter−eff
ets des corrections de liaisons
      /// </summary>
      /// <param name="joint1">Première liaison (i)</param>
      /// <param name="joint2">Seconde liaison (j)</param>
      /// <returns>Matrice Mij relative aux 2 liaisons</returns>
      public static Mapack.Matrix Mij(BaseJoint joint1, BaseJoint joint2)
      {
         if (joint1.Solid1 == joint2.Solid1)
         {
            if (joint1.Solid2 != joint2.Solid2)
            {
               return Nij(joint1, joint2, joint1.Solid1, joint1.Solid1Point, joi
nt2.Solid1Point);
            }
            else
            {
               return (Nij(joint1, joint2, joint1.Solid1, joint1.Solid1Point, jo
int2.Solid1Point)
                        + Nij(joint1, joint2, joint1.Solid2, joint1.Solid2Point,
 joint2.Solid2Point));
            }
         }
```

```
         if (joint1.Solid1 != joint2.Solid1 && joint1.Solid2 == joint2.Solid2)
         {
            return Nij(joint1, joint2, joint1.Solid2, joint1.Solid2Point, joint2
.Solid2Point);
         }

         if (joint1.Solid1 == joint2.Solid2)
         {
            if (joint1.Solid2 != joint2.Solid1)
            {
               return -Nij(joint1, joint2, joint1.Solid1, joint1.Solid1Point, jo
int2.Solid2Point);
            }
            else
            {
               return -(Nij(joint1, joint2, joint1.Solid1, joint1.Solid1Point, j
oint2.Solid2Point)
                        + Nij(joint1, joint2, joint1.Solid2, joint1.Solid2Point,
 joint2.Solid1Point));
            }
         }

         if (joint1.Solid1 != joint2.Solid2 && joint1.Solid2 == joint2.Solid1)
         {
            return -Nij(joint1, joint2, joint1.Solid2, joint1.Solid2Point, joint
2.Solid1Point);
         }

         return new Mapack.Matrix(3, 3);
      }

      /// <summary>
      /// Détermine la matrice du système de liaisons dans l'état courant des so
lides
      /// </summary>
      /// <param name="joints">Collection contenant les liaisons du système</par
am>
      /// <returns>Matrice représentative du système</returns>
      public static Mapack.Matrix SystemMatrix(List<BaseJoint> joints)
      {
         // On vérifie que la collection contient bien des liaisons
         if (joints.Count == 0)
            throw new Exception("Pas de liaison dans la collection");

         // On détermine la taille de la matrice et on crée celle-ci
         int dimension = 0;
         foreach (BaseJoint joint in joints)
         {
            dimension += joint.NumberOfConstraints;
         }
         Mapack.Matrix system = new Mapack.Matrix(dimension, dimension);


         // On remplit la matrice
         Mapack.Matrix subMatrix;
         int row = 0, column = 0;
         // On remplit ligne par ligne
         for (int i = 0; i < joints.Count; i++)
         {
            for (int j = 0; j < joints.Count; j++)
            {
               // On définit la sous-matrice correspondant aux 2 liaisons
               subMatrix = Mij(joints[i], joints[j]);
               subMatrix = joints[i].ProjectionMatrix * (subMatrix * (joints[j].
ProjectionMatrix.Transpose()));
               MatrixHelper.SetSubMatrix(system, subMatrix, row, column);
```

```
               // On incrémente le décalage de colonnes
               column += joints[j].NumberOfConstraints;
            }
            // On incrémente le décalage de lignes
            row += joints[i].NumberOfConstraints;
            // On repart de la première colonne
            column = 0;
         }

         return system;
      }

      /// <summary>
      /// Détermine le second membre permettant de corriger les erreurs de posit
ionnement des liaisons
      /// </summary>
      /// <param name="joints">Collection contenant les liaisons du système</par
am>
      /// <returns>Second membre du système</returns>
      public static Mapack.Matrix JointCorrectionSecondMember(List<BaseJoint> jo
ints, double deltatime)
      {
         // On vérifie que la collection contient bien des liaisons
         if (joints.Count == 0)
            throw new Exception("Pas de liaison dans la collection");

         // On détermine la taille de la matrice et on crée celle-ci
         int dimension = 0;
         foreach (BaseJoint joint in joints)
         {
            dimension += joint.NumberOfConstraints;
         }
         Mapack.Matrix secondMember = new Mapack.Matrix(dimension, 3);

         // On remplit la matrice
         int row = 0;
         Mapack.Matrix subMatrix;
         for (int i = 0; i < joints.Count; i++)
         {
            // On ajoute la partie correspondante à la liaison
            subMatrix = joints[i].ProjectionMatrix * (joints[i].GetJointError()
* (1/deltatime));
            MatrixHelper.SetSubMatrix(secondMember, subMatrix, row, 0);

            // On incrémente le décalage de lignes
            row += joints[i].NumberOfConstraints;
         }

         return secondMember;
      }

      /// <summary>
      /// Détermine le second membre permettant de corriger les erreurs de vites
se des liaisons
      /// </summary>
      /// <param name="joints">Collection contenant les liaisons du système</par
am>
      /// <returns>Second membre du système</returns>
      public static Mapack.Matrix JointSpeedCorrectionSecondMember(List<BaseJoin
t> joints)
      {
         // On vérifie que la collection contient bien des liaisons
         if (joints.Count == 0)
            throw new Exception("Pas de liaison dans la collection");
```

```
         // On détermine la taille de la matrice et on crée celle-ci
         int dimension = 0;
         foreach (BaseJoint joint in joints)
         {
            dimension += joint.NumberOfConstraints;
         }
         Mapack.Matrix secondMember = new Mapack.Matrix(dimension, 3);

         // On remplit la matrice
         int row = 0;
         Mapack.Matrix subMatrix;
         for (int i = 0; i < joints.Count; i++)
         {
            // On ajoute la partie correspondante à la liaison
            subMatrix = joints[i].ProjectionMatrix * joints[i].GetJointSpeedErro
r();
            MatrixHelper.SetSubMatrix(secondMember, subMatrix, row, 0);

            // On incrémente le décalage de lignes
            row += joints[i].NumberOfConstraints;
         }

         return secondMember;
      }

      #endregion

      #region Résolution

      /// <summary>
      /// Effectue une itération de correction de position des liaisons
      /// </summary>
      /// <param name="deltatime">Durée du pas de calcul de la frame</param>
      public void ResolutionStep(double deltatime)
      {
         // La matrice du système a été calculée à l'étape précédente
         // On la calcule si l'on se trouve à la première utilisation
         if (system == null)
         {
            system = SystemMatrix(assemblyObject.AssemblyJoints);
            inversedSystem = system.Inverse;
         }

         // On récupère le second membre
         Mapack.Matrix secondMember = JointCorrectionSecondMember(assemblyObject
.AssemblyJoints, deltatime);

         // On en déduit les corrections à appliquer
         Mapack.Matrix impulses = inversedSystem * secondMember;

         // On récupère les impulsions ou moments angulaires à appliquer et on l
es applique
         int row = 0;
         BaseJoint currentJoint;
         Mapack.Matrix subMatrix;
         Vector3 correction;
         for (int i = 0; i < assemblyObject.AssemblyJoints.Count; i++)
         {
            currentJoint = assemblyObject.AssemblyJoints[i];
            // On récupère la projection de la correction
            subMatrix = impulses.Submatrix(row, row + currentJoint.NumberOfConst
raints - 1, 0, 2);
            // On détermine la correction
            correction = MatrixHelper.Vector3FromMatrix((currentJoint.Projection
Matrix.Transpose() * subMatrix).Submatrix(0, 2, 0, 0));
```

```
            // On applique la correction selon le type de liaison
            if (currentJoint.Type == BaseJoint.JointType.TranslationalConstraint
)
            {
               // Ici la correction est une impulsion
               //Solide 1
               currentJoint.Solid1.ApplyImpulseAtPoint(-correction, currentJoint
.Solid1ApplicationPoint);
               //Solide 2
               currentJoint.Solid2.ApplyImpulseAtPoint(correction, currentJoint.
Solid2ApplicationPoint);

               //currentJoint.Solid1.ApplyImpulseAtPoint(correction, currentJoin
t.Solid1Point);
               //currentJoint.Solid2.ApplyImpulseAtPoint(-correction, currentJoi
nt.Solid2Point);
            }
            else
            {
               // Ici la correction est un moment angulaire
               currentJoint.Solid1.ApplyAngularMomentum(-correction);
               currentJoint.Solid2.ApplyAngularMomentum(correction);
            }

            // On incrémente le décalage de lignes
            row += currentJoint.NumberOfConstraints;
         }
      }

      /// <summary>
      /// Effectue l'étape de correction de vitesse dans les liaisons
      /// Cette étape calcule la matrice à l'étape suivante. Elle doit donc clor
e la résolution
      /// </summary>
      public void SpeedCorrectionStep()
      {
         // On calcule la matrice du système correspondant au nouvel état de l'a
ssemblage
         system = SystemMatrix(assemblyObject.AssemblyJoints);
         inversedSystem = system.Inverse;

         // On récupère le second membre
         Mapack.Matrix secondMember = JointSpeedCorrectionSecondMember(assemblyO
bject.AssemblyJoints);

         // On en déduit les corrections à appliquer
         Mapack.Matrix impulses = inversedSystem * secondMember;

         // On récupère les impulsions ou moments angulaires à appliquer et on l
es applique
         int row = 0;
         BaseJoint currentJoint;
         Mapack.Matrix subMatrix;
         Vector3 correction;
         for (int i = 0; i < assemblyObject.AssemblyJoints.Count; i++)
         {
            currentJoint = assemblyObject.AssemblyJoints[i];
            // On récupère la projection de la correction
            subMatrix = impulses.Submatrix(row, row + currentJoint.NumberOfConst
raints - 1, 0, 2);
            // On détermine la correction
            correction = MatrixHelper.Vector3FromMatrix((currentJoint.Projection
Matrix.Transpose() * subMatrix).Submatrix(0, 2, 0, 0));

            // On applique la correction selon le type de liaison
            if (currentJoint.Type == BaseJoint.JointType.TranslationalConstraint
```

```
)
            {
               // Ici la correction est une impulsion
               currentJoint.Solid1.ApplyImpulseAtPoint(-correction, currentJoint
.Solid1Point);
               currentJoint.Solid2.ApplyImpulseAtPoint(correction, currentJoint.
Solid2Point);
            }
            else
            {
               // Ici la correction est un moment angulaire
               currentJoint.Solid1.ApplyAngularMomentum(-correction);
               currentJoint.Solid2.ApplyAngularMomentum(correction);
            }

            // On incrémente le décalage de lignes
            row += currentJoint.NumberOfConstraints;
         }
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;

namespace Kinema
{
   public class MatrixHelper
   {
      #region Transformations Vecteurs-Matrices

      /// <summary>
      /// Transforme le vecteur en une matrice 2x1
      /// </summary>
      /// <param name="vector">Le vecteur à transformer</param>
      /// <returns>La matrice colonne du vecteur</returns>
      public static Mapack.Matrix MatrixFromVector2(Vector2 vector)
      {
         Mapack.Matrix matrix = new Mapack.Matrix(2, 1);
         matrix[0, 0] = vector.X;
         matrix[1, 0] = vector.Y;

         return matrix;
      }

      /// <summary>
      /// Transforme le vecteur en une matrice 3x1
      /// </summary>
      /// <param name="vector">Le vecteur à transformer</param>
      /// <returns>La matrice colonne du vecteur</returns>
      public static Mapack.Matrix MatrixFromVector3(Vector3 vector)
      {
         Mapack.Matrix matrix = new Mapack.Matrix(3, 1);
         matrix[0, 0] = vector.X;
         matrix[1, 0] = vector.Y;
         matrix[2, 0] = vector.Z;

         return matrix;
      }

      /// <summary>
      /// Transforme le vecteur en une matrice 4x1
      /// </summary>
      /// <param name="vector">Le vecteur à transformer</param>
      /// <returns>La matrice colonne du vecteur</returns>
      public static Mapack.Matrix MatrixFromVector4(Vector4 vector)
      {
         Mapack.Matrix matrix = new Mapack.Matrix(4, 1);
         matrix[0, 0] = vector.X;
         matrix[1, 0] = vector.Y;
         matrix[2, 0] = vector.Z;
         matrix[3, 0] = vector.W;

         return matrix;
      }

      /// <summary>
      /// Transforme une matrice 2x1 en un vecteur à 2 dimensions
      /// </summary>
      /// <param name="matrix">La matrice colonne du vecteur</param>
      /// <returns>Le vecteur correspondant à la matrice</returns>
      public static Vector2 Vector2FromMatrix(Mapack.Matrix matrix)
      {
         if (matrix.Columns != 1 || matrix.Rows != 2)
            throw new Exception("La matrice n’a pas le format d’un vecteur à 2 d
imensions.");
```

```
         Vector2 vecteur = Vector2.Empty;

         vecteur.X = (float)matrix[0, 0];
         vecteur.Y = (float)matrix[1, 0];

         return vecteur;
      }

      /// <summary>
      /// Transforme une matrice 3x1 en un vecteur à 3 dimensions
      /// </summary>
      /// <param name="matrix">La matrice colonne du vecteur</param>
      /// <returns>Le vecteur correspondant à la matrice</returns>
      public static Vector3 Vector3FromMatrix(Mapack.Matrix matrix)
      {
         if (matrix.Columns != 1 || matrix.Rows != 3)
            throw new Exception("La matrice n’a pas le format d’un vecteur à 3 d
imensions.");

         Vector3 vecteur = Vector3.Empty;

         vecteur.X = (float)matrix[0, 0];
         vecteur.Y = (float)matrix[1, 0];
         vecteur.Z = (float)matrix[2, 0];

         return vecteur;
      }

      /// <summary>
      /// Transforme une matrice 4x1 en un vecteur à 4 dimensions
      /// </summary>
      /// <param name="matrix">La matrice colonne du vecteur</param>
      /// <returns>Le vecteur correspondant à la matrice</returns>
      public static Vector4 Vector4FromMatrix(Mapack.Matrix matrix)
      {
         if (matrix.Columns != 1 || matrix.Rows != 4)
            throw new Exception("La matrice n’a pas le format d’un vecteur à 4 d
imensions.");

         Vector4 vecteur = Vector4.Empty;

         vecteur.X = (float)matrix[0, 0];
         vecteur.Y = (float)matrix[1, 0];
         vecteur.Z = (float)matrix[2, 0];
         vecteur.W = (float)matrix[3, 0];

         return vecteur;
      }

      #endregion

      #region Transformations entre formats de Matrices

      /// <summary>
      /// Transforme une matrice directX 4x4 en une matrice Mapack 3x3
      /// </summary>
      /// <param name="matrix">Matrice DirectX 4x4 </param>
      /// <returns>Matrice Mapack 3x3 provenant du bloc supérieur gauche</return
s>
      public static Mapack.Matrix DirectXToMapackMatrix33(Matrix matrix)
      {
         Mapack.Matrix mat = new Mapack.Matrix(3, 3);

         mat[0, 0] = matrix.M11;
         mat[1, 0] = matrix.M21;
```

```
         mat[2, 0] = matrix.M31;
         mat[0, 1] = matrix.M12;
         mat[1, 1] = matrix.M22;
         mat[2, 1] = matrix.M32;
         mat[0, 2] = matrix.M13;
         mat[1, 2] = matrix.M23;
         mat[2, 2] = matrix.M33;

         return mat;
      }

      /// <summary>
      /// Transforme une matrice directX 4x4 en une matrice Mapack 4x4
      /// </summary>
      /// <param name="matrix">Matrice DirectX 4x4 </param>
      /// <returns>Matrice Mapack 4x4</returns>
      public static Mapack.Matrix DirectXToMapackMatrix44(Matrix matrix)
      {
         Mapack.Matrix mat = new Mapack.Matrix(4, 4);

         mat[0, 0] = matrix.M11;
         mat[1, 0] = matrix.M21;
         mat[2, 0] = matrix.M31;
         mat[3, 0] = matrix.M41;
         mat[0, 1] = matrix.M12;
         mat[1, 1] = matrix.M22;
         mat[2, 1] = matrix.M32;
         mat[3, 1] = matrix.M42;
         mat[0, 2] = matrix.M13;
         mat[1, 2] = matrix.M23;
         mat[2, 2] = matrix.M33;
         mat[3, 2] = matrix.M43;
         mat[0, 3] = matrix.M14;
         mat[1, 3] = matrix.M24;
         mat[2, 3] = matrix.M34;
         mat[3, 3] = matrix.M44;

         return mat;
      }

      /// <summary>
      /// Transforme une matrix Mapack 3x3 en une matrice directX 4x4
      /// </summary>
      /// <param name="matrix">Matrice Mapack 3x3 à transformer</param>
      /// <returns>Matrice directX</returns>
      public static Matrix MapackToDirectXMatrix33(Mapack.Matrix matrix)
      {
         if (matrix.Columns != 3 || matrix.Rows != 3)
            throw new Exception("La matrice Mapack n'est pas une matrice 3x3");

         Matrix mat = new Matrix();

         mat.M11 = (float)matrix[0, 0];
         mat.M21 = (float)matrix[1, 0];
         mat.M31 = (float)matrix[2, 0];
         mat.M12 = (float)matrix[0, 1];
         mat.M22 = (float)matrix[1, 1];
         mat.M32 = (float)matrix[2, 1];
         mat.M13 = (float)matrix[0, 2];
         mat.M23 = (float)matrix[1, 2];
         mat.M33 = (float)matrix[2, 2];

         return mat;
      }

      /// <summary>
```

```
      /// Transforme une matrix Mapack 4x4 en une matrice directX 4x4
      /// </summary>
      /// <param name="matrix">Matrice Mapack 4x4 à transformer</param>
      /// <returns>Matrice directX</returns>
      public static Matrix MapackToDirectXMatrix44(Mapack.Matrix matrix)
      {
         if (matrix.Columns != 4 || matrix.Rows != 4)
            throw new Exception("La matrice Mapack n'est pas une matrice 4x4");

         Matrix mat = new Matrix();

         mat.M11 = (float)matrix[0, 0];
         mat.M21 = (float)matrix[1, 0];
         mat.M31 = (float)matrix[2, 0];
         mat.M41 = (float)matrix[3, 0];
         mat.M12 = (float)matrix[0, 1];
         mat.M22 = (float)matrix[1, 1];
         mat.M32 = (float)matrix[2, 1];
         mat.M42 = (float)matrix[3, 1];
         mat.M13 = (float)matrix[0, 2];
         mat.M23 = (float)matrix[1, 2];
         mat.M33 = (float)matrix[2, 2];
         mat.M43 = (float)matrix[3, 2];
         mat.M14 = (float)matrix[0, 3];
         mat.M24 = (float)matrix[1, 3];
         mat.M34 = (float)matrix[2, 3];
         mat.M44 = (float)matrix[3, 3];

         return mat;
      }

      #endregion

      #region Opérations sur les matrices Mapack

      /// <summary>
      /// Définit une partie d'une matrice à partir de la sous-matrice donnée
      /// </summary>
      /// <param name="matrix">Matrice à modifier</param>
      /// <param name="subMatrix">Matrice contenant les valeurs à insérer</param
>
      /// <param name="row">Ligne de début de la sous-matrice</param>
      /// <param name="column">Colonne de début de la sous-matrice</param>
      public static void SetSubMatrix(Mapack.Matrix matrix, Mapack.Matrix subMat
rix, int row, int column)
      {
         if ((row + subMatrix.Rows > matrix.Rows) || (column + subMatrix.Columns
 > matrix.Columns))
            throw new Exception("La matrice ne peut pas accueillir cette sous-ma
trice à la position indiquée");

         for (int i = 0; i < subMatrix.Rows; i++)
            for (int j = 0; j < subMatrix.Columns; j++)
            {
               matrix[row + i, column + j] = subMatrix[i, j];
            }
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe gérant une Boite englobante dont les cotés sont alignés avec les a
xes
   /// </summary>
   public class BoundingBox
   {
      #region Variables de la classe

      protected Vector3 minimalPoint = new Vector3(), maximalPoint = new Vector3
();

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur par défaut
      /// </summary>
      public BoundingBox() : this(Vector3.Empty, Vector3.Empty) { }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="minimalPoint">Point du pavé dont les coordonnées sont min
imales</param>
      /// <param name="maximalPoint">Point du pavé dont les coordonnées sont max
imales</param>
      public BoundingBox(Vector3 minimalPoint, Vector3 maximalPoint)
      {
         MinimalPoint = minimalPoint;
         MaximalPoint = maximalPoint;

         if (!IsValid)
            throw new Exception("Pavé invalide");
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Point du pavé dont les coordonnées sont minimales
      /// </summary>
      public Vector3 MinimalPoint
      {
         get
         {
            return minimalPoint;
         }
         set
         {
            minimalPoint.X = value.X;
            minimalPoint.Y = value.Y;
            minimalPoint.Z = value.Z;
         }
      }

      /// <summary>
```

```
      /// Point du pavé dont les coordonnées sont maximales
      /// </summary>
      public Vector3 MaximalPoint
      {
         get
         {
            return maximalPoint;
         }
         set
         {
            maximalPoint.X = value.X;
            maximalPoint.Y = value.Y;
            maximalPoint.Z = value.Z;
         }
      }

      /// <summary>
      /// Détermine si le pavé est un cube
      /// </summary>
      public bool IsACube
      {
         get
         {
            if (!IsValid)
               throw new Exception("Pavé invalide");

            Vector3 diag = MaximalPoint - MinimalPoint;
            if (diag.X == diag.Y && diag.X == diag.Z)
               return true;
            return false;
         }
      }

      /// <summary>
      /// Détermine si les données du pavé sont valides
      /// </summary>
      public bool IsValid
      {
         get
         {
            if (MaximalPoint.X < MinimalPoint.X || MaximalPoint.Y < MinimalPoint
.Y || MaximalPoint.Z < MinimalPoint.Z)
               return false;
            return true;
         }
      }

      /// <summary>
      /// Calcule le rayon de la sphere circonscrite au cube
      /// Sinon génère une exception
      /// </summary>
      public float EnglobingRadius
      {
         get
         {
            return (Vector3.Length(MaximalPoint - MinimalPoint) / 2.0f);
         }
      }

      /// <summary>
      /// Calcule le center du pavé
      /// </summary>
      public Vector3 Center
      {
         get
         {
```

```
            return 0.5f * (MaximalPoint + MinimalPoint);
         }
      }

      #endregion

      #region Méthodes de génération

      /// <summary>
      /// Détermine la boite englobante d'un ensemble de triangles
      /// </summary>
      /// <param name="triangles">Ensemble de triangles à englober</param>
      /// <returns>Boite englobant les triangles</returns>
      public static BoundingBox GenerateFromTriangles(List<Triangle> triangles)
      {
         if (triangles.Count == 0)
            throw new Exception("Aucun triangle dans la collection");

         // Initialisation
         Vector3 minimalPoint = triangles[0][0].Position;
         Vector3 maximalPoint = triangles[0][0].Position;

         // On parcourt l'ensemble des points à la recherche des extrema
         for (int i=0; i < triangles.Count; i++)
            for (int j = 0; j < 3; j++)
            {
               minimalPoint = Vector3.Minimize(minimalPoint, triangles[i][j].Pos
ition);
               maximalPoint = Vector3.Maximize(maximalPoint, triangles[i][j].Pos
ition);
            }

         return new BoundingBox(minimalPoint, maximalPoint);
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Transforme le pavé en son plus petit cube englobant de même centre
      /// </summary>
      public void MakeCubic()
      {
         if (!IsValid)
            throw new Exception("Pavé invalide");

         Vector3 diag = MaximalPoint - MinimalPoint;
         Vector3 center = Center;

         float dist = 0.5f * Math.Max(diag.X, Math.Max(diag.Y, diag.Z));

         Vector3 newHalfDiag = new Vector3(dist, dist, dist);

         MinimalPoint = center - newHalfDiag;
         MaximalPoint = center + newHalfDiag;
      }

      /// <summary>
      /// Détermine si un point se trouve à l'intérieur de la boite
      /// </summary>
      /// <param name="point">Point à tester</param>
      /// <returns><see langword="true"/> si le point est à l'intérieur de la bo
ite</returns>
      public bool IsPointInside(Vector3 point)
      {
```

```
         return ((point.X >= MinimalPoint.X) && (point.X <= MaximalPoint.X) &&
                 (point.Y >= MinimalPoint.Y) && (point.Y <= MaximalPoint.Y) &&
                 (point.Z >= MinimalPoint.Z) && (point.Z <= MaximalPoint.Z));
      }

      /// <summary>
      /// Détermine si des points de la collection se trouvent à l'intérieur de
la boite
      /// </summary>
      /// <param name="points">Collection de points à tester</param>
      /// <returns><see langword="true"/> si au moins un point de la collection
est à l'intérieur de la boite</returns>
      public bool HasPointInside(List<Vector3> points)
      {
         foreach (Vector3 point in points)
         {
            if (IsPointInside(point))
               return true;
         }

         return false;
      }

      /// <summary>
      /// Détermine les points de la collection situés à l'intérieur du pavé
      /// (cette fonciton n'élimine pas les doublons présents dans la collection
 initial)
      /// </summary>
      /// <param name="points">Collection de points à tester</param>
      /// <returns>Points de la collection se trouvant strictement à l'intérieur
 de la boite</returns>
      public List<Vector3> InsidePoints(List<Vector3> points)
      {
         List<Vector3> insidePoints = new List<Vector3>();

         foreach (Vector3 point in points)
         {
            if (IsPointInside(point))
               insidePoints.Add(point);
         }

         return insidePoints;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using System.IO;

namespace Kinema
{
   /// <summary>
   /// Gère un arbre de sphere pour la detections de collisions
   /// </summary>
   public class BoundingSphereTree
   {
      #region Variables de la classe

      BoundingSphereTreeNode head;
      Matrix worldMatrix;
      float scalingValue;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur par défaut
      /// </summary>
      public BoundingSphereTree() : this(null) { }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="headSphere">Racine de l'arbre</param>
      public BoundingSphereTree(BoundingSphereTreeNode headSphere)
      {
         head = headSphere;
      }

      #endregion

      #region Construction des arbres

      /// <summary>
      /// Génère un octree de spheres à partir d'un Mesh
      /// </summary>
      /// <param name="mesh">Mesh dont il faut générer l'enveloppe</param>
      /// <param name="level">Niveau de division voulu dans l'arbre</param>
      /// <param name="samplePointsPrecision">Precision de la génération du nuag
e de points représentant l'objet</param>
      public void GenerateOctreeFromMesh(Mesh mesh, int level, int samplePointsP
recision)
      {
         GenerateOctreeFromMesh(mesh, level, samplePointsPrecision, false);
      }

      /// <summary>
      /// Génère un octree de spheres à partir d'un Mesh
      /// </summary>
      /// <param name="mesh">Mesh dont il faut générer l'enveloppe</param>
      /// <param name="level">Niveau de division voulu dans l'arbre</param>
      /// <param name="samplePointsPrecision">Precision de la génération du nuag
e de points représentant l'objet</param>
      /// <param name="writingInfosToFile">Si <see langword="true"/>, écrit la l
iste des vertices dans un fichier</param>
      public void GenerateOctreeFromMesh(Mesh mesh, int level, int samplePointsP
```

```
recision, bool writingInfosToFile)
      {
         // On vérifie la validité du niveau de division demandé
         if (level <= 0)
            throw new Exception("Niveau de division de l'octree invalide");

         // On récupère la liste des triangles constituant l'objet
         List<Triangle> triangles = GetTrianglesFromMesh(mesh, writingInfosToFil
e);

         // On récupère le nuage de points représentant l'objet
         List<Vector3> samplePoints = GetSamplePoints(triangles, samplePointsPre
cision);

         // On génère la racine de l'arbre
         BoundingBox box = BoundingBox.GenerateFromTriangles(triangles);
         box.MakeCubic();
         head = new BoundingSphereTreeNode(box.Center, box.EnglobingRadius);

         // On génère le reste de l'arbre
         DivideBoundingCube(head, box, samplePoints, level, 1);
      }

      /// <summary>
      /// Génère un arbre binaire à partir d'un Mesh
      /// </summary>
      /// <param name="mesh">Mesh dont il faut générer l'enveloppe</param>
      public void GenerateBinaryTreeFromMesh(Mesh mesh)
      {
         GenerateBinaryTreeFromMesh(mesh, false);
      }

      /// <summary>
      /// Génère un arbre binaire à partir d'un Mesh
      /// </summary>
      /// <param name="mesh">Mesh dont il faut générer l'enveloppe</param>
      /// <param name="writingInfosToFile">Si <see langword="true"/>, écrit la l
iste des vertices dans un fichier</param>
      public void GenerateBinaryTreeFromMesh(Mesh mesh, bool writingInfosToFile)
      {
         // On construit l'arbre de manière down-top
         // On stocke la liste des triangles dans une collection
         List<Triangle> triangleList = GetTrianglesFromMesh(mesh);

         // On assigne une sphere à chaque triangle et l'on stocke le tout dans
une collection
         List<BoundingSphereTreeNode> boundingSphereList = new List<BoundingSphe
reTreeNode>();
         BoundingSphereTreeNode node;

         foreach (Triangle triangle in triangleList)
         {
            node = new BoundingSphereTreeNode();
            node.TriangleList.Add(triangle);
            node.GenerateFromTriangleList();
            boundingSphereList.Add(node);
         }

         // On construit l'arbre en regroupant progressivement le spheres
         while (boundingSphereList.Count > 1)
         {
            boundingSphereList = ReduceSphereLevel(boundingSphereList);
         }

         // On a réduit l'arbre à une seule sphère, la tête.
         head = boundingSphereList[0];
```

```
      }

      #endregion

      #region Calculs d'intersections

      /// <summary>
      /// Détermine l'intersection d'un rayon et de l'arbre
      /// </summary>
      /// <param name="rayPosition">Point d'origine du rayon</param>
      /// <param name="rayDirection">Direction du rayon</param>
      /// <param name="intersectedNode">Renvoie la feuille traversée par le rayo
n la plus proche de son origine</param>
      /// <returns><see langword="true"/> si le rayon traverse l'intérieur d'une
 feuille de l'arbre</returns>
      public bool Intersect(Vector3 rayPosition, Vector3 rayDirection, out Bound
ingSphereTreeNode intersectedNode)
      {
         return head.Intersect(rayPosition, rayDirection, out intersectedNode, w
orldMatrix, scalingValue);
      }

      /// <summary>
      /// Détermine les paires de spheres des arbres qui s'intersectent
      /// </summary>
      /// <param name="tree1">Premier arbre</param>
      /// <param name="tree2">Second arbre</param>
      /// <param name="intersectedPairsFrom1">Renvoie les spheres du premier arb
re</param>
      /// <param name="intersectedPairsFrom2">Renvoie les spheres du second arbr
e</param>
      /// <returns>Nombre de paires de spheres qui s'intersectent</returns>
      public static int Intersect(BoundingSphereTree tree1, BoundingSphereTree t
ree2,
         out ArrayList intersectedPairsFrom1, out ArrayList intersectedPairsFrom
2)
      {
         intersectedPairsFrom1 = new ArrayList();
         intersectedPairsFrom2 = new ArrayList();

         BoundingSphereTree.IntersectNodes(tree1.head, tree2.head, tree1.worldMa
trix, tree2.worldMatrix,
            tree1.scalingValue, tree2.scalingValue, intersectedPairsFrom1, inter
sectedPairsFrom2);

         return intersectedPairsFrom1.Count;
      }

      #endregion

      #region Affichage de l'arbre

      /// <summary>
      /// Dessine un niveau de l'arbre de spheres
      /// </summary>
      /// <param name="device">device sur lequel effectuer le rendu</param>
      /// <param name="level">Niveau de l'arbre à dessine, -1 dessine toutes les
 feuilles</param>
      public void Render(Device device, int level)
      {
         // On vérifie la validité du niveau demandé
         if (level < -1 || level == 0) return;

         // On crée le Mesh et le material des spheres
         Mesh sphere = Mesh.Sphere(device, 1.0f, 20, 20);
         Material material = new Material();
```

```
         material.Diffuse = System.Drawing.Color.FromArgb(10, 255, 0, 0);
         material.Ambient = System.Drawing.Color.FromArgb(10, 255, 0, 0);
         material.Specular = System.Drawing.Color.FromArgb(10, 255, 0, 0);
         material.Emissive = System.Drawing.Color.FromArgb(10, 255, 0, 0);

         // On active l'alpha blending basé sur les materials et on définit le m
aterial
         device.RenderState.DiffuseMaterialSource = ColorSource.Material;
         device.Material = material;
         device.SetTexture(0, null);

         // On parcourt l'arbre à la recherche des sphères du niveau demandé
         head.Render(device, worldMatrix, sphere, level, 1);

         // On n'a plus besoin du Mesh
         sphere.Dispose();
      }

      /// <summary>
      /// Dessine les paires de spheres des 2 arbres qui sont en collision
      /// </summary>
      /// <param name="device">Device sur lequel effectuer le rendu</param>
      /// <param name="tree1">Premier arbre</param>
      /// <param name="tree2">Second arbre</param>
      public static void RenderCollision(Device device, BoundingSphereTree tree1
, BoundingSphereTree tree2)
      {
         ArrayList spheres1, spheres2;

         if (BoundingSphereTree.Intersect(tree1, tree2, out spheres1, out sphere
s2) == 0)
            return;

         // On crée le Mesh et le material des spheres
         Mesh sphere = Mesh.Sphere(device, 1.0f, 20, 20);
         Material material = new Material();
         material.Diffuse = System.Drawing.Color.FromArgb(10, 0, 0, 255);
         material.Ambient = System.Drawing.Color.FromArgb(10, 0, 0, 255);
         material.Specular = System.Drawing.Color.FromArgb(10, 0, 0, 255);
         material.Emissive = System.Drawing.Color.FromArgb(10, 0, 0, 255);

         // On active l'alpha blending basé sur les materials et on définit le m
aterial
         device.RenderState.DiffuseMaterialSource = ColorSource.Material;
         device.Material = material;
         device.SetTexture(0, null);

         // On dessine les spheres
         for (int i = 0; i < spheres1.Count; i++)
         {
            ((BoundingSphereTreeNode)spheres1[i]).Render(device, tree1.worldMatr
ix, sphere);
            ((BoundingSphereTreeNode)spheres2[i]).Render(device, tree2.worldMatr
ix, sphere);
         }

         sphere.Dispose();
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Met à jour la transformation monde de l'objet représenté par l'arbre
      /// </summary>
```

```
      /// <param name="world">Matrice de transformation monde</param>
      /// <param name="scalingFactor">Facteur de mise à l'échelle du modèle</par
am>
      public void UpdateWorldMatrix(Matrix world, float scalingFactor)
      {
         worldMatrix = world;
         scalingValue = scalingFactor;
      }

      /// <summary>
      /// Regroupe les spheres d'une collection en paires pour construire un arb
re binaire
      /// </summary>
      /// <param name="boundingSphereList">Liste de spheres à réduire</param>
      /// <returns>Liste réduite</returns>
      protected List<BoundingSphereTreeNode> ReduceSphereLevel(List<BoundingSphe
reTreeNode> boundingSphereList)
      {
         List<BoundingSphereTreeNode> newBoundingSphereList = new List<BoundingS
phereTreeNode>();
         List<BoundingSphereTreeNode> sphereList = new List<BoundingSphereTreeNo
de>(boundingSphereList);

         BoundingSphereTreeNode closestSphere, currentSphere, newSphere;
         float squaredDistance = 0f, testDistance;

         // On s'arrange pour avoir un nombre pair de spheres dans le niveau à t
raiter
         if (sphereList.Count / 2 != sphereList.Count / 2.0f)
         {
            newBoundingSphereList.Add(sphereList[sphereList.Count - 1]);
            sphereList.RemoveAt(sphereList.Count - 1);
         }

         // On regroupe les spheres
         while (sphereList.Count > 1)
         {
            currentSphere = sphereList[0];
            closestSphere = sphereList[1];
            squaredDistance = Vector3.LengthSq(closestSphere.Center - currentSph
ere.Center);

            for (int i = 2; i < sphereList.Count; i++)
            {
               testDistance = Vector3.LengthSq(sphereList[i].Center - currentSph
ere.Center);

               if (testDistance < squaredDistance)
               {
                  squaredDistance = testDistance;
                  closestSphere = sphereList[i];
               }
            }

            // On a trouvé la sphere la plus proche de currentSphere
            // On regroupe ces 2 spheres
            newSphere = new BoundingSphereTreeNode();
            newSphere.TriangleList.AddRange(currentSphere.TriangleList);
            newSphere.TriangleList.AddRange(closestSphere.TriangleList);
            newSphere.GenerateFromTriangleList();
            newSphere.Children.Add(currentSphere);
            newSphere.Children.Add(closestSphere);

            // On ajoute la nouvelle sphere au niveau et on efface les 2 ancienn
es
            sphereList.Remove(currentSphere);
```

```
            sphereList.Remove(closestSphere);
            newBoundingSphereList.Add(newSphere);
         }

         // On a regroupé toutes les spheres, on renvoie le résultat
         return newBoundingSphereList;
      }

      /// <summary>
      /// Divise un cube en 8 cubes identiques pour propager la contruction d'un
 octree de spheres englobantes
      /// </summary>
      /// <param name="currentNode">Noeud de l'arbre à diviser</param>
      /// <param name="boundingBox">Cube initial à diviser</param>
      /// <param name="points">Liste des points à approximer par l'arbre</param>
      /// <param name="wantedLevel">Niveau de division voulu pour l'arbre final<
/param>
      /// <param name="currentLevel">Niveau de l'arbre auquel le noeud à diviser
 appartient</param>
      protected void DivideBoundingCube(BoundingSphereTreeNode currentNode, Boun
dingBox boundingBox, List<Vector3> points,
         int wantedLevel, int currentLevel)
      {
         // On vérifie si l'on a dépassé le niveu demandé
         if (currentLevel >= wantedLevel)
            return;

         // Sinon, on divise le cube
         BoundingBox box = new BoundingBox(boundingBox.MinimalPoint, boundingBox
.MaximalPoint);
         BoundingBox newBox;
         BoundingSphereTreeNode newSphere;
         List<Vector3> insidePoints;

         if (!box.IsACube)
            box.MakeCubic();
         float halfDist = 0.5f * (box.MaximalPoint.X - box.MinimalPoint.X);

         Vector3 halfX = new Vector3(halfDist, 0, 0);
         Vector3 halfY = new Vector3(0, halfDist, 0);
         Vector3 halfZ = new Vector3(0, 0, halfDist);
         Vector3 diag = new Vector3(halfDist, halfDist, halfDist);
         Vector3 minPoint;

         for (int i=0; i < 2; i++)
            for (int j=0; j < 2; j++)
               for (int k = 0; k < 2; k++)
               {
                  // On génère la nouvelle boite
                  minPoint = box.MinimalPoint + i * halfX + j * halfY + k * half
Z;
                  newBox = new BoundingBox(minPoint, minPoint + diag);

                  // Si la boite contient encore des points, on lui associe une
sphere et on poursuit la génération
                  insidePoints = newBox.InsidePoints(points);
                  if (insidePoints.Count > 0)
                  {
                     newSphere = new BoundingSphereTreeNode(newBox.Center, newBo
x.EnglobingRadius);
                     currentNode.Children.Add(newSphere);

                     // On poursuit la génération avec les points intérieur au p
avé pour accélérer
                     DivideBoundingCube(newSphere, newBox, insidePoints, wantedL
evel, currentLevel + 1);
```

```
                }
            }
        }

        /// <summary>
        /// Méthode de parcourt des arbres pour détecter les paires de spheres en
collision
        /// </summary>
        /// <param name="node1">noeud courant du premier arbre</param>
        /// <param name="node2">noeud courant du second arbre</param>
        /// <param name="world1">Matrice de transformation monde de l'objet repré
senté par le premier arbre</param>
        /// <param name="world2">Matrice de transformation monde de l'objet repré
senté par le second arbre</param>
        /// <param name="scalingFactor1">Facteur de mise à l'échelle du modèle rep
résenté par le premier arbre</param>
        /// <param name="scalingFactor2">Facteur de mise à l'échelle du modèle rep
résenté par le second arbre</param>
        /// <param name="intersectedPairsFrom1">Liste à mettre à jour des spheres
de l'arbre 1 en collision</param>
        /// <param name="intersectedPairsFrom2">Liste à mettre à jour des spheres
de l'arbre 2 en collision</param>
        protected static void IntersectNodes(BoundingSphereTreeNode node1, Boundin
gSphereTreeNode node2,
            Matrix world1, Matrix world2, float scalingFactor1, float scalingFactor
2,
            ArrayList intersectedPairsFrom1, ArrayList intersectedPairsFrom2)
        {
            // Si l'un des noeuds est à null, il n'y a pas d'intersection
            if (node1 == null || node2 == null)
                return;

            // Les noeuds existant, on les met à jour
            node1.Transform(world1, scalingFactor1);
            node2.Transform(world2, scalingFactor2);

            // Si les 2 noeuds ne s'intersectent pas, on peut répondre
            if (!node1.Intersect(node2))
                return;

            // A partir d'ici les 2 noeuds s'intersectent
            // Si ce sont des feuilles, on a trouvé une paire
            if (node1.IsLeaf && node2.IsLeaf)
            {
                intersectedPairsFrom1.Add(node1);
                intersectedPairsFrom2.Add(node2);

                return;
            }

            // On parcourt l'arbre :
            // Si un des noeud est une feuille, on parcourt l'autre branche
            // Sinon on parcourt la branche de la plus grande sphere

            // On teste les conditions de parcourt de la branche du premier noeud
            if (node2.IsLeaf || node1.Radius >= node2.Radius)
            {
                // On intersecte le noeud 2 avec les branches issues du noeud 1
                foreach (BoundingSphereTreeNode node in node1.Children)
                {
                    BoundingSphereTree.IntersectNodes(node, node2, world1, world2, sc
alingFactor1, scalingFactor2,
                    intersectedPairsFrom1, intersectedPairsFrom2);
                }

                return;
```

```
         }

         // Ici, node1 est une feuille ou la sphere de node2 est plus grosse
         // On intersecte donc le noeud 1 avec les branches issues du noeud 2
         foreach (BoundingSphereTreeNode node in node2.Children)
         {
            BoundingSphereTree.IntersectNodes(node1, node, world1, world2, scali
ngFactor1, scalingFactor2,
            intersectedPairsFrom1, intersectedPairsFrom2);
         }
      }

      /// <summary>
      /// Récupère la liste des triangles constituant un Mesh
      /// </summary>
      /// <param name="mesh">Modèle dont il faut extraire les triangles</param>
      /// <returns>Liste des triangles du modèle</returns>
      protected static List<Triangle> GetTrianglesFromMesh(Mesh mesh)
      {
         return GetTrianglesFromMesh(mesh, false);
      }

      /// <summary>
      /// Récupère la liste des triangles constituant un Mesh
      /// </summary>
      /// <param name="mesh">Modèle dont il faut extraire les triangles</param>
      /// <param name="writingInfosToFile">Si <see langword="true"/>, écrit les
vertex et indice buffers dans un fichier texte</param>
      /// <returns>Liste des triangles du modèle</returns>
      protected static List<Triangle> GetTrianglesFromMesh(Mesh mesh, bool writi
ngInfosToFile)
      {
         // On récupère la liste des vertices
         Vector3[] vertices = new Vector3[mesh.NumberVertices];

         // On récupère le format des vertices stockés dans le Mesh
         VertexElement[] meshDeclaration = mesh.Declaration;
         // On cherche l'offset auquel se trouve l'information de position
         int positionOffset = -1;
         for (int i = 0; i < meshDeclaration.Length; i++)
         {
            if (meshDeclaration[i].DeclarationUsage == DeclarationUsage.Position
 && meshDeclaration[i].UsageIndex == 0)
            {
               positionOffset = i;
               break;
            }
         }
         // On vérifie que l'on a bien trouvé la position
         if (positionOffset == -1)
            throw new Exception("Les vertices du Mesh ne contiennent pas d'infor
mations de position !");

         // On lit le vertex buffer pour récupérer les positions
         GraphicsStream vb = mesh.LockVertexBuffer(LockFlags.ReadOnly);

         for (long i = 0; i < mesh.NumberVertices; i++)
         {
            vb.Seek(i * mesh.NumberBytesPerVertex + positionOffset, SeekOrigin.B
egin);
            vertices[i] = (Vector3)vb.Read(typeof(Vector3));
         }

         // On récupère l'index buffer dans le format sous lequel il est stocké
et on le convertit si necessaire
         Int32[] indices = new Int32[mesh.NumberFaces * 3];
```

```
         if (mesh.IndexBuffer.Description.Is16BitIndices == true)
         {
            Int16[] indicesFromBuffer = (Int16[])mesh.LockIndexBuffer(typeof(Int
16), 0, mesh.NumberFaces * 3);
            for (int i = 0; i < indicesFromBuffer.Length; i++)
               indices[i] = (Int32)indicesFromBuffer[i];
         }
         else
         {
            indices = (Int32[])mesh.LockIndexBuffer(typeof(Int32), 0, mesh.Numbe
rFaces * 3);
         }

         // On écrit les vertices dans un fichier pour voir a quoi elles ressemb
lent...si on le souhaite...
         if (writingInfosToFile)
         {
            // On ouvre le fichier en écriture
            System.IO.FileStream output = new System.IO.FileStream("vertices.txt
", FileMode.Append, FileAccess.Write);
            System.IO.StreamWriter fileWriter = new System.IO.StreamWriter(outpu
t);
            for (int i = 0; i < vertices.Length; i++)
            {
               fileWriter.WriteLine("vertex n°" + (i + 1) + "\t: " + vertices[i]
.X + "\t\t" + vertices[i].Y + "\t\t" + vertices[i].Z);
            }
            fileWriter.WriteLine("");

            for (int i = 0; i < indices.Length; i++)
            {
               fileWriter.WriteLine("index n°" + (i + 1) + "\t: " + (indices[i]
+ 1));
            }
            fileWriter.WriteLine("==========================================
==================\n");
            fileWriter.Close();
            output.Close();
         }

         // On récupère la liste des triangles
         List<Triangle> triangleList = new List<Triangle>();
         Triangle meshTriangle;

         for (int i = 0; i < mesh.NumberFaces; i++)
         {
            meshTriangle = new Triangle();
            meshTriangle[0] = new CustomVertex.PositionOnly(vertices[indices[3 *
 i]]);
            meshTriangle[1] = new CustomVertex.PositionOnly(vertices[indices[3 *
 i + 1]]);
            meshTriangle[2] = new CustomVertex.PositionOnly(vertices[indices[3 *
 i + 2]]);

            triangleList.Add(meshTriangle);
         }

         // On unlock le vertex buffer et l'index buffer
         mesh.UnlockVertexBuffer();
         mesh.UnlockIndexBuffer();

         return triangleList;
      }

      /// <summary>
      /// Récupère une liste de points contenus dans les triangles pour générer
```

```
un nuage représentant l'objet
      /// </summary>
      /// <param name="triangles">Liste des triangles constituant l'objet</param
>
      /// <param name="precision">Nombre d'intervalles découpés dans les cotés,
precision*(precision+1)/2 points générés</param>
      /// <returns>Collection de points représentant l'objet</returns>
      protected static List<Vector3> GetSamplePoints(List<Triangle> triangles, i
nt precision)
      {
         List<Vector3> points = new List<Vector3>();
         Vector3 point;

         float f, g;
         int k;

         foreach (Triangle triangle in triangles)
         {
            k = precision;
            f = 0;

            for (int i = 0; i <= precision; i++)
            {
               g = 0;
               for (int j = 0; j <= k; j++)
               {
                  point = Vector3.BaryCentric(triangle[0].Position, triangle[1].
Position, triangle[2].Position, f, g);
                  if (!points.Contains(point))
                     points.Add(point);

                  g += 1 / (float)precision;
               }
               k--;
               f += 1 / (float)precision;
            }
         }

         return points;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Représente un noeud ou une feuille d'un arbre de spheres englobantes
   /// </summary>
   public class BoundingSphereTreeNode
   {
      #region Variables de la classe

      protected List<BoundingSphereTreeNode> children;
      protected Vector3 center = new Vector3(), transformedCenter = new Vector3(
);
      protected float radius, transformedRadius;
      protected List<Triangle> triangleList;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur par défaut
      /// </summary>
      public BoundingSphereTreeNode()
         : this(Vector3.Empty, 0.0f, new List<BoundingSphereTreeNode>()) { }

      /// <summary>
      /// Construit une feuille de l'arbre
      /// </summary>
      /// <param name="sphereCenter">Centre de la sphere</param>
      /// <param name="sphereRadius">Rayon de la sphere</param>
      public BoundingSphereTreeNode(Vector3 sphereCenter, float sphereRadius)
         : this(sphereCenter, sphereRadius, new List<BoundingSphereTreeNode>())
{ }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="sphereCenter">Centre de la sphere</param>
      /// <param name="sphereRadius">Rayon de la sphere</param>
      /// <param name="leaves">Liste des enfants du noeud</param>
      public BoundingSphereTreeNode(Vector3 sphereCenter, float sphereRadius,
         List<BoundingSphereTreeNode> children)
      {
         TriangleList = new List<Triangle>();
         Children = new List<BoundingSphereTreeNode>(children);
         Center = sphereCenter;
         Radius = sphereRadius;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Rayon de la sphere dans l'espace modèle
      /// </summary>
      public float Radius
      {
         get
         {
```

```
            return radius;
         }
         set
         {
            radius = value;
         }
      }

      /// <summary>
      /// Rayon de la sphere dans l'espace monde
      /// </summary>
      public float TransformedRadius
      {
         get
         {
            return transformedRadius;
         }
         set
         {
            transformedRadius = value;
         }
      }

      /// <summary>
      /// Centre de la sphere dont l'espace modèle
      /// </summary>
      public Vector3 Center
      {
         get
         {
            return center;
         }
         set
         {
            center.X = value.X;
            center.Y = value.Y;
            center.Z = value.Z;
         }
      }

      /// <summary>
      /// Centre de la sphere dans l'espace monde
      /// </summary>
      public Vector3 TransformedCenter
      {
         get
         {
            return transformedCenter;
         }
         set
         {
            transformedCenter.X = value.X;
            transformedCenter.Y = value.Y;
            transformedCenter.Z = value.Z;
         }
      }

      /// <summary>
      /// Liste des enfants de la sphere
      /// </summary>
      public List<BoundingSphereTreeNode> Children
      {
         get
         {
            return children;
         }
```

```
         protected set
         {
            children = value;
         }
      }

      /// <summary>
      /// Liste des triangles contenus dans la sphere
      /// </summary>
      public List<Triangle> TriangleList
      {
         get
         {
            return triangleList;
         }
         protected set
         {
            triangleList = value;
         }
      }

      /// <summary>
      /// Renvoie <see langword="true"/> si le noeud est une feuille
      /// </summary>
      public bool IsLeaf
      {
         get
         {
            return (Children.Count == 0);
         }
      }

      #endregion

      #region Calculs d'intersections

      /// <summary>
      /// Détermine si un point est à l'intérieur de la sphere
      /// </summary>
      /// <param name="point">Point à tester en coordonnées objet (sans mise à l
'échelle du modèle)</param>
      /// <returns><see langword="true"/> si le point est strictement à l'intéri
eur de la sphere</returns>
      public bool IsModelPointInside(Vector3 point)
      {
         if (Vector3.LengthSq(point - Center) < Radius * Radius)
            return true;

         return false;
      }

      /// <summary>
      /// Détermine si un point est à l'intérieur de la sphere.
      /// La sphere doit avoir été mise à jour
      /// </summary>
      /// <param name="point">Point à tester en coordonnées monde</param>
      /// <returns><see langword="true"/> si le point est strictement à l'intéri
eur de la sphere</returns>
      public bool IsWorldPointInside(Vector3 point)
      {
         if (Vector3.LengthSq(point - TransformedCenter) < TransformedRadius * T
ransformedRadius)
            return true;

         return false;
      }
```

```
      /// <summary>
      /// Détermine si la sphere est intersectée par celle passée en paramètre
      /// </summary>
      /// <param name="sphere">Sphere à tester</param>
      /// <param name="worldMatrix">Matrice de transformation monde</param>
      /// <param name="scalingValue">Facteur de mise à l'échelle du modèle</para
m>
      /// <returns><see langword="true"/> si les 2 spheres s'intersectent strict
ement</returns>
      public bool Intersect(BoundingSphereTreeNode sphere, Matrix worldMatrix, f
loat scalingValue)
      {
         Transform(worldMatrix, scalingValue);
         sphere.Transform(worldMatrix, scalingValue);

         return Intersect(sphere);
      }

      /// <summary>
      /// Détermine si la sphere est intersectée par celle passée en paramètre
      /// Les 2 spheres doivent avoir été transformées avant l'appel à cette fon
ction
      /// </summary>
      /// <param name="sphere">Sphere à tester</param>
      /// <returns><see langword="true"/> si les 2 spheres s'intersectent strict
ement</returns>
      public bool Intersect(BoundingSphereTreeNode sphere)
      {
         return (Vector3.LengthSq(sphere.TransformedCenter − TransformedCenter)
            < ((TransformedRadius + sphere.TransformedRadius) * (TransformedRadi
us + sphere.TransformedRadius)));
      }

      /// <summary>
      /// Détermine si un rayon traverse l'intérieur d'une feuille de l'arbre
      /// </summary>
      /// <param name="rayPosition">Point d'origine du rayon</param>
      /// <param name="rayDirection">Direction du rayon</param>
      /// <param name="intersectedNode">Renvoie la feuille traversée par le rayo
n la plus proche de son origine</param>
      /// <param name="worldMatrix">Matrice de transformation monde</param>
      /// <param name="scalingvalue">Facteur de mise à l'échelle du modèle</para
m>
      /// <returns><see langword="true"/> si le rayon traverse l'intérieur de la
 sphere</returns>
      public bool Intersect(Vector3 rayPosition, Vector3 rayDirection,
         out BoundingSphereTreeNode intersectedNode, Matrix worldMatrix, float s
calingvalue)
      {
         bool result;

         // Test avec la sphere courante
         Transform(worldMatrix, scalingvalue);
         result = Geometry.SphereBoundProbe(TransformedCenter, TransformedRadius
, rayPosition, rayDirection);

         // S'il n'y a pas d'intersection
         if (!result)
         {
            intersectedNode = null;
            return false;
         }

         // Si c'est une feuille, on a trouvé ce que l'on cherche
         if (IsLeaf)
```

```
         {
            intersectedNode = this;

            return result;
         }

         // Sinon on continue à parcourir l'arbre
         List<BoundingSphereTreeNode> intersected = new List<BoundingSphereTreeN
ode>();
         BoundingSphereTreeNode intersectedChild = null;

         // On parcourt l'arbre
         foreach (BoundingSphereTreeNode node in Children)
         {
            node.Intersect(rayPosition, rayDirection, out intersectedChild, worl
dMatrix, scalingvalue);

            if (intersectedChild != null)
            {
               intersected.Add(intersectedChild);
               intersectedChild = null;
            }
         }

         // S'il n'y a pas d'intersection, on peut répondre
         if (intersected.Count == 0)
         {
            intersectedNode = null;
            return false;
         }

         // Sinon on cherche la sphere intersectée la plus proche de l'origine d
u rayon
         BoundingSphereTreeNode closestIntersectedChild = intersected[0];
         float squaredClosestDistance = Vector3.LengthSq(intersected[0].Transfor
medCenter - rayPosition);
         float squaredDistance;

         foreach (BoundingSphereTreeNode node in intersected)
         {
            squaredDistance = Vector3.LengthSq(node.TransformedCenter - rayPosit
ion);

            if (squaredDistance < squaredClosestDistance)
            {
               squaredClosestDistance = squaredDistance;
               closestIntersectedChild = node;
            }
         }

         intersectedNode = closestIntersectedChild;
         return true;
      }

      #endregion

      #region Affichage de l'arbre

      // Méthode permettant d'affiche un niveau de l'arbre
      // Le niveau -1 correspond à l'ensemble des feuilles
      /// <summary>
      /// Dessine récursivement un niveau de l'arbre
      /// </summary>
      /// <param name="device">Device sur lequel effectuer le rendu</param>
      /// <param name="worldMatrix">Matrice de transformation monde</param>
      /// <param name="model">Mesh dont le subset 0 sera dessiné</param>
```

```
      /// <param name="wantedLevel">Niveau de l'arbre à dessiner, -1 dessine l'e
nsemble des feuilles</param>
      /// <param name="currentLevel">Niveau auquel le noeud appartient</param>
      public virtual void Render(Device device, Matrix worldMatrix, Mesh model,
int wantedLevel, int currentLevel)
      {
         // Si l'on se trouve à un niveau inférieur au niveau voulu on arrête de
 parcourir l'arbre
         if (wantedLevel < currentLevel && wantedLevel != -1)
            return;

         // Si l'on se trouve au niveau voulu, on dessine le modèle
         if ((wantedLevel == currentLevel) || (wantedLevel == -1 && IsLeaf))
         {
            Render(device, worldMatrix, model);
            return;
         }

         // On parcourt l'arbre
         foreach (BoundingSphereTreeNode node in Children)
         {
            node.Render(device, worldMatrix, model, wantedLevel, currentLevel +
1);
         }
      }

      /// <summary>
      /// Dessine la sphere sur le device
      /// </summary>
      /// <param name="device">Device sur lequel effectuer le rendu</param>
      /// <param name="worldMatrix">Matrice de transformation monde</param>
      /// <param name="model">Mesh dont le subset 0 sera dessiné</param>
      public virtual void Render(Device device, Matrix worldMatrix, Mesh model)
      {
         // On crée la matrice monde correcte pour la sphere courante
         //Matrix world = Matrix.Multiply(Matrix.Translation(Center), worldMatri
x);
         //world = Matrix.Multiply(Matrix.Scaling(Radius, Radius, Radius), world
);
         Matrix world = new Matrix();
         world.AffineTransformation(Radius, Vector3.Empty, Quaternion.Identity,
Center);
         world = Matrix.Multiply(world, worldMatrix);
         device.Transform.World = world;

         model.DrawSubset(0);
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Génère la sphere à partir de la liste de triangles qu'elle stocke
      /// </summary>
      public void GenerateFromTriangleList()
      {
         // Version donnant une sphere englobante presque optimale
         /*float rad, radSquared, xspan, yspan, zspan, maxspan;
         float oldToCurrent, oldToCurrentSquared, oldToNew;
         Vector3 xmin, xmax, ymin, ymax, zmin, zmax, dia1, dia2, cen;

         // On récupère un tableau contenant les points
         Vector3[] points = new Vector3[TriangleList.Count * 3];

         for (int i = 0; i < TriangleList.Count; i++)
```

```
        {
           points[3 * i] = TriangleList[i][0].Position;
           points[3 * i + 1] = TriangleList[i][1].Position;
           points[3 * i + 2] = TriangleList[i][2].Position;
        }

        // Première passe, on cherche les 6 points extremes
        // On prend les valeurs du premier point pour l'ensemble
        xmin = points[0]; xmax = points[0];
        ymin = points[0]; ymax = points[0];
        zmin = points[0]; zmax = points[0];

        for (int i = 0; i < points.Length; i++)
             {
           if (points[i].X < xmin.X)
              xmin = points[i]; // nouveau point de coordonnée x minimale
           if (points[i].X > xmax.X)
              xmax = points[i];
           if (points[i].Y < ymin.Y)
              ymin = points[i];
           if (points[i].Y > ymax.Y)
              ymax = points[i];
           if (points[i].Z < zmin.Z)
              zmin = points[i];
           if (points[i].Z > zmax.Z)
              zmax = points[i];
                }
        // On calcule la deistance séparant les 2 points extrêmes pour x, y et
z
        xspan = Vector3.LengthSq(xmax − xmin);
        yspan = Vector3.LengthSq(ymax − ymin);
        zspan = Vector3.LengthSq(zmax − zmin);

        // On cherche la paire la plus espacée pour obtenir un premier diamètre
        dia1 = xmin; dia2 = xmax;
        maxspan = xspan;
        if (yspan > maxspan)
             {
                maxspan = yspan;
               dia1 = ymin; dia2 = ymax;
             }
        if (zspan > maxspan)
             {
                dia1 = zmin; dia2 = zmax;
             }

        // [dia1,dia2] est un diamètre de la sphere initiale
        // On calcule le centre
        cen = 1 / 2.0f * (dia1 + dia2);
        // Puis le rayon et son carré
        radSquared = Vector3.LengthSq(dia2 − cen);
        rad = (float)Math.Sqrt(radSquared);

        // Seconde passe, on affine la sphere obtenue

        for (int i = 0; i < points.Length; i++)
             {
           oldToCurrentSquared = Vector3.LengthSq(points[i] − cen);
                if (oldToCurrentSquared > radSquared)  // On teste d'abord le r
ayon au carré
                     {
               // Ce point est en dehors de la sphere
                         oldToCurrent = (float)Math.Sqrt(oldToCurrentSquared);
                         // On calcule le rayon de la nouvelle sphere
                         rad = (rad + oldToCurrent) / 2.0f;
                         radSquared = rad*rad;  // Pour le prochain test sur le
```

```
carré du rayon
                        oldToNew = oldToCurrent − rad;
                        // On calcule le centre de la nouvelle sphere
                cen = 1 / oldToCurrent * (rad * cen + oldToNew * points[i]);
                      }
            }

            // On stocke les informations obtenues
            Radius = rad;
            Center = cen;*/

            // Version utilisant la fonction de directX
            /*CustomVertex.PositionOnly[] verticesList = new CustomVertex.PositionO
nly[TriangleList.Count * 3];

            for (int i = 0; i < TriangleList.Count; i++)
            {
               verticesList[3 * i] = TriangleList[i][0];
               verticesList[3 * i + 1] = TriangleList[i][1];
               verticesList[3 * i + 2] = TriangleList[i][2];
            }

            Radius = Geometry.ComputeBoundingSphere(verticesList, CustomVertex.Posi
tionOnly.Format, out center);*/

            // On détermine le barycentre de la distribution          ...ce que fai
t DirectX...
            float x = 0.0f, y = 0.0f, z = 0.0f;

            for (int i = 0; i < TriangleList.Count; i++)
            {
               x += TriangleList[i].Barycentre.X;
               y += TriangleList[i].Barycentre.Y;
               z += TriangleList[i].Barycentre.Z;
            }

            Vector3 barycenter = new Vector3(x, y, z);
            Center = (1 / (float)triangleList.Count) * barycenter;
            //Center = new Vector3(20, 20, 20);

            // On cherche le point le plus éloigné du barycentre pour obtenir le ra
yon de la sphere
            float longerDistance = Vector3.LengthSq(TriangleList[0][0].Position − C
enter);
            float squaredDistance = 0.0f;

            for (int i = 0; i < TriangleList.Count; i++)
               for (int j = 0; j < 3; j++)
               {
                  squaredDistance = Vector3.LengthSq(TriangleList[i][j].Position −
Center);

                  if (squaredDistance > longerDistance)
                     longerDistance = squaredDistance;
               }

            Radius = (float)Math.Sqrt(longerDistance);
         }

         /// <summary>
         /// Détermine si certains des points de la collection se trouvent à l'inté
rieur de la sphere
         /// </summary>
         /// <param name="points">Collection de points à tester en coordonnées obje
t (sans mise à l'échelle du modèle)</param>
         /// <returns><see langword="true"/> si un des points est strictement à l'i
```

```
ntérieur de la sphere</returns>
      public bool HasModelPointsInside(List<Vector3> points)
      {
         // On teste chaque point
         foreach (Vector3 point in points)
         {
            // Si le point est à l'intérieur, on renvoie vrai
            if (IsModelPointInside(point))
               return true;
         }

         // Ici, aucun point de la collection n'est à l'intérieur de la sphere,
donc :
         return false;
      }

      /// <summary>
      /// Détermine si certains des points de la collection se trouvent à l'inté
rieur de la sphere
      /// </summary>
      /// <param name="points">Collection de points à tester en coordonnées mond
e</param>
      /// <returns><see langword="true"/> si un des points est strictement à l'i
ntérieur de la sphere</returns>
      public bool HasWorldPointsInside(List<Vector3> points)
      {
         // On teste chaque point
         foreach (Vector3 point in points)
         {
            // Si le point est à l'intérieur, on renvoie vrai
            if (IsWorldPointInside(point))
               return true;
         }

         // Ici, aucun point de la collection n'est à l'intérieur de la sphere,
donc :
         return false;
      }

      // Projette la sphère dans les coordonnées monde
      /// <summary>
      /// Projette la sphere dans l'espace monde
      /// </summary>
      /// <param name="worldMatrix">Matrice de transformation monde</param>
      /// <param name="scalingValue">Facteur de mise à l'échelle du modèle</para
m>
      public void Transform(Matrix worldMatrix, float scalingValue)
      {
         TransformedCenter = Vector3.TransformCoordinate(Center, worldMatrix);
         TransformedRadius = Radius * scalingValue;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Représente un triangle en 3 dimensions permettant de manipuler les Mesh
   /// </summary>
   public class Triangle
   {
      #region Variables de la classe

      protected CustomVertex.PositionOnly[] points = new CustomVertex.PositionOn
ly[3];
      protected Vector3 barycentre = new Vector3();
      protected bool isBarycentreCalculated = false;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur par défaut : crée un triangle dont les 3 sommets sont à l
'origine du repère
      /// </summary>
      public Triangle() : this(Vector3.Empty, Vector3.Empty, Vector3.Empty) { }

      /// <summary>
      /// Crée un triangle à partir de 3 points
      /// </summary>
      /// <param name="A">Premier sommet</param>
      /// <param name="B">Deuxième sommet</param>
      /// <param name="C">Troisième sommet</param>
      public Triangle(Vector3 A, Vector3 B, Vector3 C)
      {
         points[0] = new CustomVertex.PositionOnly(A);
         points[1] = new CustomVertex.PositionOnly(B);
         points[2] = new CustomVertex.PositionOnly(C);
      }

      /// <summary>
      /// Crée un triangle à partir de 3 Vertices
      /// </summary>
      /// <param name="A">Premier sommet</param>
      /// <param name="B">Deuxième sommet</param>
      /// <param name="C">Troisième sommet</param>
      public Triangle(CustomVertex.PositionOnly A, CustomVertex.PositionOnly B,
CustomVertex.PositionOnly C)
      {
         points[0] = A;
         points[1] = B;
         points[2] = C;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Indexeur permettant de récupérer la liste des Vertices constituant le
triangle
      /// </summary>
      /// <param name="index">Numéro du point dans [0,2]</param>
```

```
      /// <returns>Vertex définissant le point</returns>
      public CustomVertex.PositionOnly this[int index]
      {
         get
         {
            return points[index];
         }
         set
         {
            points[index] = value;
            isBarycentreCalculated = false;
         }
      }

      /// <summary>
      /// Renvoie le barycentre du triangle
      /// </summary>
      public Vector3 Barycentre
      {
         get
         {
            if (!isBarycentreCalculated) CalculateBarycentre();

            return barycentre;
         }
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Détermine le barycentre du triangle
      /// </summary>
      protected void CalculateBarycentre()
      {
         barycentre.X = (points[0].X + points[1].X + points[2].X) / 3.0f;
         barycentre.Y = (points[0].Y + points[1].Y + points[2].Y) / 3.0f;
         barycentre.Z = (points[0].Z + points[1].Z + points[2].Z) / 3.0f;

         isBarycentreCalculated = true;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   public class BallJoint : BaseJoint
   {
      #region Variables de la classe

      protected static Mapack.Matrix projectionMatrix = new Mapack.Matrix(3, 3,
1);
      protected float precision, squaredPrecision;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="solid1">Premier solide de la liaison</param>
      /// <param name="solid2">Second solide de la liaison</param>
      /// <param name="solid1Point">Point de la liaison sur le solide 1 en coord
onnées objet</param>
      /// <param name="solid2Point">Point de la liaison sur le solide 2 en coord
onnées objet</param>
      /// <param name="precision">Ecart de position autorisé au niveau de la lia
ison</param>
      public BallJoint(KinemaBaseObject solid1, KinemaBaseObject solid2, Vector3
 solid1Point, Vector3 solid2Point, float precision)
         : base(solid1, solid2, JointType.TranslationalConstraint, solid1Point,
solid2Point)
      {
         Precision = precision;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Nombre de degrés de libertés contraints par la liaison
      /// </summary>
      public override int NumberOfConstraints
      {
         get
         {
            return 3;
         }
      }

      /// <summary>
      /// Matrice de projection de la liaison
      /// </summary>
      public override Mapack.Matrix ProjectionMatrix
      {
         get
         {
            return projectionMatrix;
         }
         protected set
         {
         }
```

```
      }

      /// <summary>
      /// Ecart de positionnement accepté sur la liaison
      /// </summary>
      public float Precision
      {
         get
         {
            return precision;
         }
         set
         {
            precision = value;
            squaredPrecision = value * value;
         }
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Détermine si les contraintes de liaisons sont satisfaites
      /// </summary>
      /// <returns><see langword="true"/> si les contraintes sont satisfaites</r
eturns>
      public override bool IsConstraintSatified()
      {
         if (Vector3.LengthSq(Solid1Point − Solid2Point) <= squaredPrecision)
         {
            return true;
         }
         else
         {
            return false;
         }
      }

      /// <summary>
      /// Calcule l’erreur de positionnement de la liaison
      /// </summary>
      /// <returns>Erreur de positionnement de la liaison</returns>
      public override Mapack.Matrix GetJointError()
      {
         return MatrixHelper.MatrixFromVector3(Solid1Point − Solid2Point);
      }

      /// <summary>
      /// Calcule l’erreur de vitesse de la liaison
      /// </summary>
      /// <returns>Erreur de vitesse de la liaison</returns>
      public override Mapack.Matrix GetJointSpeedError()
      {
         return MatrixHelper.MatrixFromVector3(Solid1.NextState.PointSpeed(Solid
1Point) − Solid2.NextState.PointSpeed(Solid2Point));
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   public class BallSlider : BaseJoint
   {
      #region Variables de la classe

      protected Mapack.Matrix projectionMatrix = new Mapack.Matrix(2, 3);
      protected float precision, squaredPrecision;
      protected Vector3 directionObject = new Vector3(), directionWorld = new Ve
ctor3();
      protected Vector3[] planeBase = new Vector3[2], planeWorldBase = new Vecto
r3[2];

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur complet.
      /// Le degré de liberté autorisé est défini en coordonnées objet du solide
 1 par solid1Point + k*direction
      /// </summary>
      /// <param name="solid1">Premier solide de la liaison</param>
      /// <param name="solid2">Second solide de la liaison</param>
      /// <param name="solid1Point">Point de la liaison sur le solide 1 en coord
onnées objet</param>
      /// <param name="solid2Point">Point de la liaison sur le solide 2 en coord
onnées objet</param>
      /// <param name="direction">Direction du degré de liberté autorisé en coor
données objet du solide 1</param>
      /// <param name="precision">Ecart de position autorisé au niveau de la lia
ison</param>
      public BallSlider(KinemaBaseObject solid1, KinemaBaseObject solid2, Vector
3 solid1Point, Vector3 solid2Point, Vector3 direction, float precision)
         : base(solid1, solid2, JointType.TranslationalConstraint, solid1Point,
solid2Point)
      {
         PlaneBase[0] = new Vector3();
         PlaneBase[1] = new Vector3();
         PlaneWorldBase[0] = new Vector3();
         PlaneWorldBase[1] = new Vector3();

         Precision = precision;
         DirectionObject = direction;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Nombre de degrés de libertés contraints par la liaison
      /// </summary>
      public override int NumberOfConstraints
      {
         get
         {
            return 2;
         }
      }
```

```
      /// <summary>
      /// Matrice de projection de la liaison
      /// </summary>
      public override Mapack.Matrix ProjectionMatrix
      {
         get
         {
            return projectionMatrix;
         }
         protected set
         {
            projectionMatrix = value;
         }
      }

      /// <summary>
      /// Ecart de positionnement accepté sur la liaison
      /// </summary>
      public float Precision
      {
         get
         {
            return precision;
         }
         set
         {
            precision = value;
            squaredPrecision = value * value;
         }
      }

      /// <summary>
      /// Direction du degré de liberté de la liaison en coordonnées objet du so
lide 1
      /// </summary>
      public Vector3 DirectionObject
      {
         get
         {
            return directionObject;
         }
         set
         {
            directionObject.X = value.X;
            directionObject.Y = value.Y;
            directionObject.Z = value.Z;

            // On génère le plan de contraintes
            GenerateConstraintPlane(value);
         }
      }

      /// <summary>
      /// Direction du degré de liberté de la liaison en coordonnées monde
      /// </summary>
      public Vector3 DirectionWorld
      {
         get
         {
            return directionWorld;
         }
         set
         {
            directionWorld.X = value.X;
            directionWorld.Y = value.Y;
```

```
            directionWorld.Z = value.Z;
         }
      }

      /// <summary>
      /// Base du plan normal au DDL autorisé (en coordonnées objet)
      /// </summary>
      protected Vector3[] PlaneBase
      {
         get
         {
            return planeBase;
         }
      }

      /// <summary>
      /// Base du plan normal au DDL autorisé (en coordonnées monde)
      /// </summary>
      protected Vector3[] PlaneWorldBase
      {
         get
         {
            return planeWorldBase;
         }
      }

      /// <summary>
      /// Point du solide 1 auquel les impulsions de correction doivent être app
liquées
      /// </summary>
      public override Vector3 Solid1ApplicationPoint
      {
         get
         {
            // On doit calculer le point de l'axe du DDL le plus proche de l'axe
 du solide 2
            // Et ce dans la position courante de l'objet
            // (la mise à jour a fait le calcul dans la position suivante)

            // On calcule les coordonnées monde des points des solides ainsi que
 la direction du DDL
            Vector3 solid1point = Vector3.TransformCoordinate(Solid1ObjectPoint,
 Solid1.WorldMatrix);
            Vector3 solid2point = Vector3.TransformCoordinate(Solid2ObjectPoint,
 Solid2.WorldMatrix);
            Vector3 direction = Vector3.TransformNormal(DirectionObject, Solid1.
WorldMatrix);
            direction.Normalize();

            // On rapproche le point le long du DDL le plus près possible du poi
nt du solide 2 le long du DDL autorisé
            float offset = Vector3.Dot(solid2point - solid1point, direction);
            return solid1point + offset * direction;
         }
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Met à jour les positions des points et la matrice de projection de la
liaison
      /// </summary>
      /// <param name="solid1WorldMatrix">Matrice de transformation monde du sol
ide 1</param>
```

```
      /// <param name="solid2WorldMatrix">Matrice de transformation monde du sol
ide 2</param>
      public override void UpdateParameters(Matrix solid1WorldMatrix, Matrix sol
id2WorldMatrix)
      {
         // On met à jour les positions des points
         base.UpdateParameters(solid1WorldMatrix, solid2WorldMatrix);

         // On projette dans l'espace monde la direction du DDL
         DirectionWorld = Vector3.TransformNormal(DirectionObject, solid1WorldMa
trix);
         DirectionWorld.Normalize();

         // On génère une base du plan de contrainte dans l'espace monde
         PlaneWorldBase[0] = Vector3.TransformNormal(PlaneBase[0], solid1WorldMa
trix);
         PlaneWorldBase[0].Normalize();
         PlaneWorldBase[1] = Vector3.TransformNormal(PlaneBase[1], solid1WorldMa
trix);
         PlaneWorldBase[1].Normalize();

         // On calcule la matrice de projection de la liaison
         MatrixHelper.SetSubMatrix(ProjectionMatrix, MatrixHelper.MatrixFromVect
or3(PlaneWorldBase[0]).Transpose(), 0, 0);
         MatrixHelper.SetSubMatrix(ProjectionMatrix, MatrixHelper.MatrixFromVect
or3(PlaneWorldBase[1]).Transpose(), 1, 0);

         // On rapproche le point de la liaison du solide 1 au plus près de celu
i du solide 2 le long du DDL autorisé
         float offset = Vector3.Dot(Solid2Point − Solid1Point, DirectionWorld);
         Solid1Point = Solid1Point + offset * DirectionWorld;
      }

      /// <summary>
      /// Détermine si les contraintes de liaisons sont satisfaites
      /// </summary>
      /// <returns><see langword="true"/> si les contraintes sont satisfaites</r
eturns>
      public override bool IsConstraintSatified()
      {
         // On calcule la distance projetée
         Mapack.Matrix distanceMatrix = ProjectionMatrix * MatrixHelper.MatrixFr
omVector3(Solid2Point − Solid1Point);
         Vector2 dist = MatrixHelper.Vector2FromMatrix(distanceMatrix.Submatrix(
0, 1, 0, 0));

         if (dist.Length() <= squaredPrecision)
         {
            return true;
         }
         else
         {
            return false;
         }
      }

      /// <summary>
      /// Calcule l'erreur de positionnement de la liaison
      /// </summary>
      /// <returns>Erreur de positionnement de la liaison</returns>
      public override Mapack.Matrix GetJointError()
      {
         return MatrixHelper.MatrixFromVector3(Solid1Point − Solid2Point);
      }

      /// <summary>
```

```
      /// Calcule l'erreur de vitesse de la liaison
      /// </summary>
      /// <returns>Erreur de vitesse de la liaison</returns>
      public override Mapack.Matrix GetJointSpeedError()
      {
         return MatrixHelper.MatrixFromVector3( Solid1.NextState.PointSpeed(Soli
d1Point) − Solid2.NextState.PointSpeed(Solid2Point));
      }

      /// <summary>
      /// Génère une base d'un plan dont la normale est donnée
      /// </summary>
      /// <param name="normal"></param>
      protected void GenerateConstraintPlane(Vector3 normal)
      {
         Plane plane = Plane.FromPointNormal(Vector3.Empty, normal);
         Vector3 u = new Vector3(), v = new Vector3();
         if (plane.A != 0)
         {
            u.Y = 1;
            u.Z = 1;
            u.X = −(plane.B * u.Y + plane.C * u.Z) / plane.A;
         }
         else
         {
            // plane.A == 0
            if (plane.B != 0)
            {
               u.X = 0;
               u.Z = 1;
               u.Y = −(plane.C * u.Z) / plane.B;
            }
            else
            {
               // plane.A == 0 && plane.B == 0
               if (plane.C != 0)
               {
                  u.X = 1;
                  u.Y = 0;
                  u.Z = 0;
               }
               else
               {
                  throw new Exception("La direction du DDL autorisé n'est pas va
lide.");
               }
            }
         }
         u.Normalize();
         v = Vector3.Cross(DirectionObject, u);
         v.Normalize();

         PlaneBase[0] = u;
         PlaneBase[1] = v;
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe abstraite implémentant la logique de base d'une liaison entre 2 so
lides
   /// </summary>
   public abstract class BaseJoint
   {
      #region Variables de la classe

      protected KinemaBaseObject solid1, solid2;
      protected Vector3 solid1Point = new Vector3(), solid2Point = new Vector3()
;
      protected Vector3 solid1ObjectPoint = new Vector3(), solid2ObjectPoint = n
ew Vector3();
      protected JointType type;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="solid1">Premier solide de la liaison</param>
      /// <param name="solid2">Second solide de la liaison</param>
      /// <param name="type">Type de liaison</param>
      /// <param name="solide1Point">Point de la liaison sur le solide 1 en coor
données objet (pour les liaisons contraignant les translations)</param>
      /// <param name="solide2Point">Point de la liaison sur le solide 2 en coor
données objet (pour les liaisons contraignant les translations)</param>
      public BaseJoint(KinemaBaseObject solid1, KinemaBaseObject solid2, JointTy
pe type, Vector3 solid1Point, Vector3 solid2Point)
      {
         Solid1 = solid1;
         Solid2 = solid2;
         Type = type;
         Solid1ObjectPoint = solid1Point;
         Solid2ObjectPoint = solid2Point;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Type de contrainte exercée par la liaison
      /// </summary>
      public JointType Type
      {
         get
         {
            return type;
         }
         set
         {
            type = value;
         }
      }
```

```
      /// <summary>
      /// Nombre de degrés de libertés contraints par la liaison
      /// </summary>
      public abstract int NumberOfConstraints
      {
         get;
      }

      /// <summary>
      /// Premier solide de la liaison
      /// </summary>
      public KinemaBaseObject Solid1
      {
         get
         {
            return solid1;
         }
         set
         {
            solid1 = value;
         }
      }

      /// <summary>
      /// Second solide de la liaison
      /// </summary>
      public KinemaBaseObject Solid2
      {
         get
         {
            return solid2;
         }
         set
         {
            solid2 = value;
         }
      }

      /// <summary>
      /// Point de la liaison situé sur le solide 1 en coordonnées monde (pour l
es liaisons contraignant les translations)
      /// </summary>
      public Vector3 Solid1Point
      {
         get
         {
            return solid1Point;
         }
         set
         {
            solid1Point.X = value.X;
            solid1Point.Y = value.Y;
            solid1Point.Z = value.Z;
         }
      }

      /// <summary>
      /// Point de la liaison situé sur le solide 2 en coordonnées monde (pour l
es liaisons contraignant les translations)
      /// </summary>
      public Vector3 Solid2Point
      {
         get
         {
            return solid2Point;
         }
```

```
         set
         {
            solid2Point.X = value.X;
            solid2Point.Y = value.Y;
            solid2Point.Z = value.Z;
         }
      }

      /// <summary>
      /// Point de la liaison situé sur le solide 1 en coordonnées objet(pour le
s liaisons contraignant les translations)
      /// </summary>
      public Vector3 Solid1ObjectPoint
      {
         get
         {
            return solid1ObjectPoint;
         }
         set
         {
            solid1ObjectPoint.X = value.X;
            solid1ObjectPoint.Y = value.Y;
            solid1ObjectPoint.Z = value.Z;
         }
      }

      /// <summary>
      /// Point de la liaison situé sur le solide 2 en coordonnées objet(pour le
s liaisons contraignant les translations)
      /// </summary>
      public Vector3 Solid2ObjectPoint
      {
         get
         {
            return solid2ObjectPoint;
         }
         set
         {
            solid2ObjectPoint.X = value.X;
            solid2ObjectPoint.Y = value.Y;
            solid2ObjectPoint.Z = value.Z;
         }
      }

      /// <summary>
      /// Matrice de projection de la liaison
      /// </summary>
      public abstract Mapack.Matrix ProjectionMatrix
      {
         get;
         protected set;
      }

      /// <summary>
      /// Point du solide 1 auquel les impulsions de correction doivent être app
liquées
      /// </summary>
      public virtual Vector3 Solid1ApplicationPoint
      {
         get
         {
            return Vector3.TransformCoordinate(Solid1ObjectPoint, Solid1.WorldMa
trix);
         }
      }
```

```
      /// <summary>
      /// Point du solide 2 auquel les impulsions de correction doivent être app
liquées
      /// </summary>
      public virtual Vector3 Solid2ApplicationPoint
      {
         get
         {
            return Vector3.TransformCoordinate(Solid2ObjectPoint, Solid2.WorldMa
trix);
         }
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Détermine si les contraintes de liaisons sont satisfaites
      /// </summary>
      /// <returns><see langword="true"/> si les contraintes sont satisfaites</r
eturns>
      public abstract bool IsConstraintSatified();

      /// <summary>
      /// Met à jour les coordonnées monde des points de la liaison dans le proc
hain état des objets
      /// </summary>
      public virtual void UpdateParameters()
      {
         UpdateParameters(Solid1.NextWorldMatrix, Solid2.NextWorldMatrix);
      }

      /// <summary>
      /// Met à jour les coordonnées monde des points de la liaison
      /// </summary>
      /// <param name="worldMatrix">Matrice de transformation monde des solides<
/param>
      public virtual void UpdateParameters(Matrix solid1WorldMatrix, Matrix soli
d2WorldMatrix)
      {
         Solid1Point = Vector3.TransformCoordinate(Solid1ObjectPoint, solid1Worl
dMatrix);
         Solid2Point = Vector3.TransformCoordinate(Solid2ObjectPoint, solid2Worl
dMatrix);
      }

      /// <summary>
      /// Calcule l'erreur de positionnement de la liaison
      /// </summary>
      /// <returns>Erreur de positionnement de la liaison</returns>
      public abstract Mapack.Matrix GetJointError();

      /// <summary>
      /// Calcule l'erreur de vitesse de la liaison
      /// </summary>
      /// <returns>Erreur de vitesse de la liaison</returns>
      public abstract Mapack.Matrix GetJointSpeedError();

      #endregion

      #region Enumérations

      /// <summary>
      /// Définit les différents types de liaisons possibles
      /// </summary>
```

```
        public enum JointType {TranslationalConstraint, RotationalConstraint}

        #endregion
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   public class AssemblyObject : KinemaBaseObject
   {
      #region Variables de la classe

      protected List<KinemaBaseObject> assemblyObjects = new List<KinemaBaseObje
ct>();
      protected List<BaseJoint> assemblyJoints = new List<BaseJoint>();
      protected DynamicSolver dynamicSolver;
      protected bool isFirstResolutionStep;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur. Initialise un assemblage vide.
      /// </summary>
      public AssemblyObject() : base (null, Vector3.Empty, Quaternion.Identity,
Vector3.Empty, Vector3.Empty)
      {
         isFirstResolutionStep = true;
         dynamicSolver = new DynamicSolver(this);
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Collection contenant toutes les parties de l'assemblage
      /// </summary>
      public List<KinemaBaseObject> AssemblyObjects
      {
         get
         {
            return assemblyObjects;
         }
         set
         {
            assemblyObjects = new List<KinemaBaseObject>(value);
         }
      }

      /// <summary>
      /// Collection contenant toutes les liaisons de l'assemblage
      /// </summary>
      public List<BaseJoint> AssemblyJoints
      {
         get
         {
            return assemblyJoints;
         }
         set
         {
            assemblyJoints = new List<BaseJoint>(value);
         }
      }
```

```
        /// <summary>
        /// Détermine si les contraintes de liaisons sont satisfaites
        /// </summary>
        public override bool IsValid
        {
           get
           {
              foreach (BaseJoint joint in AssemblyJoints)
              {
                 if (!joint.IsConstraintSatified())
                    return false;
              }
              return true;
           }
        }

        #endregion

        #region Gestion du device

        /// <summary>
        /// Gère le reset du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        public override void OnDeviceReset(object sender, EventArgs e)
        {
           // On propage l'évènement vers les objets constituant l'assemblage
           foreach (KinemaBaseObject part in AssemblyObjects)
           {
              part.OnDeviceReset(sender, e);
           }
        }

        /// <summary>
        /// Gère la perte du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        public override void OnDeviceLost(object sender, EventArgs e)
        {
           // On propage l'évènement vers les objets constituant l'assemblage
           foreach (KinemaBaseObject part in AssemblyObjects)
           {
              part.OnDeviceLost(sender, e);
           }
        }

        /// <summary>
        /// Gère la libération des ressources à la destruction du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        public override void OnDeviceDisposing(object sender, EventArgs e)
        {
           // On propage l'évènement vers les objets constituant l'assemblage
           foreach (KinemaBaseObject part in AssemblyObjects)
           {
              part.OnDeviceDisposing(sender, e);
           }
        }

        #endregion

        #region Opérations de mise à jour
```

```
      /// <summary>
      /// Met à jour les parties de l'assemblage et effectue une itération de co
rrection des liaisons
      /// La mise à jour est écrite dans l'état NextState
      /// </summary>
      /// <param name="deltatime">Durée du pas de calcul de la frame</param>
      public override void Update(double deltatime)
      {
         // Si l'on effectue la première itération, on met à jour une première f
ois les objets
         if (isFirstResolutionStep)
         {
            isFirstResolutionStep = false;
            foreach (KinemaBaseObject part in AssemblyObjects)
            {
               part.Update(deltatime);
            }
            foreach (BaseJoint joint in AssemblyJoints)
            {
               joint.UpdateParameters();
            }
         }

         // On réalise une itération de la correction de position/orientation de
s liaisons
         dynamicSolver.ResolutionStep(deltatime);

         // On met à jour les objets avec prise en compte des corrections
         foreach (KinemaBaseObject part in AssemblyObjects)
         {
            part.Update(deltatime);
         }
         foreach (BaseJoint joint in AssemblyJoints)
         {
            joint.UpdateParameters();
         }
      }

      /// <summary>
      /// Ecrit l'état mis à jour dans l'état courant
      /// </summary>
      public override void FinalizeUpdate()
      {
         // On finalise les objets
         foreach (KinemaBaseObject part in AssemblyObjects)
         {
            part.FinalizeUpdate();
         }

         // On corrige les vitesses dans les liaisons
         dynamicSolver.SpeedCorrectionStep();

         // On réinitialise la résolution
         isFirstResolutionStep = true;
      }

      /// <summary>
      /// Dessine l'assemblage sur le device
      /// </summary>
      public override void Render()
      {
         // On dessine toutes les parties de l'assemblage
         foreach (KinemaBaseObject part in AssemblyObjects)
         {
            part.Render();
         }
```

```
        }

        #endregion
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe représentant un objet invisible et immobile, non soumis aux règles
 de collisions
   /// </summary>
   public class KinemaDockObject : KinemaBaseObject
   {
      #region Constructeur(s)

      /// <summary>
      /// Constructeur.
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel l'objet appartient</param>
      /// <param name="position">Position initiale du centre de masse</param>
      /// <param name="angularPosition">Position angulaire initiale, représentée
 par un quaternion</param>
      /// <param name="speed">Vitesse initiale du centre de masse</param>
      /// <param name="rotSpeed">Vitesse angulaire initiale</param>
      public KinemaDockObject(KinemaEngine kinemaEngine, Vector3 position, Quate
rnion angularPosition)
         : base(kinemaEngine, position, angularPosition, Vector3.Empty, Vector3.
Empty)
      {
         IsCollidable = false;
         IsDynamic = false;
         CalculateWorldMatrix(State);
      }

      #endregion

      #region Gestion du device

      /// <summary>
      /// Gère le reset du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceReset(object sender, EventArgs e)
      {
      }

      /// <summary>
      /// Gère la perte du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceLost(object sender, EventArgs e)
      {
      }

      /// <summary>
      /// Gère la libération des ressources à la destruction du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceDisposing(object sender, EventArgs e)
      {
      }
```

```
      #endregion

      #region Opérations de mise à jour

      /// <summary>
      /// Dessine l'objet sur le device...cet objet ne s'affiche pas
      /// </summary>
      public override void Render()
      {
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe décrivant les paramètres cinématiques d'un solide
   /// </summary>
   public class DynamicState
   {
      #region Variables de la classe

      //protected Vector3 position = new Vector3();
      protected Mapack.Matrix position, speed, angularSpeed;
      protected Quaternion angularPosition = Quaternion.Identity;
      //protected Vector3 speed = new Vector3(), angularSpeed = new Vector3();

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur par défaut
      /// </summary>
      public DynamicState() : this(new Matrix(3, 1), Quaternion.Identity, new Ma
trix(3, 1), new Matrix(3, 1)) { }

      /// <summary>
      /// Constructeur permettant d'établir la position et l'orientation de l'ob
jet
      /// </summary>
      /// <param name="position">Position de l'objet</param>
      /// <param name="angularPosition">Orientation de l'objet</param>
      public DynamicState(Matrix position, Quaternion angularPosition)
         : this(position, angularPosition, new Matrix(3, 1), new Matrix(3, 1)) {
 }

      /// <summary>
      /// Constructeur copiant un objet existant
      /// </summary>
      /// <param name="state">Objet à copier</param>
      public DynamicState(DynamicState state)
      {
         Position = state.Position;
         AngularPosition = state.AngularPosition;
         Speed = state.Speed;
         AngularSpeed = state.AngularSpeed;
      }

      /// <summary>
      /// Constructeur complet
      /// </summary>
      /// <param name="position">Position de l'objet</param>
      /// <param name="angularPosition">Orientation de l'objet</param>
      /// <param name="speed">Vitesse de l'objet</param>
      /// <param name="angularSpeed">Vecteur vitesse de rotation de l'objet</par
am>
      public DynamicState(Matrix position, Quaternion angularPosition, Matrix sp
eed, Matrix angularSpeed)
      {
         position = new Mapack.Matrix(3, 1);
         speed = new Mapack.Matrix(3, 1);
         angularSpeed = new Mapack.Matrix(3, 1);
```

```
            Position = position;
            AngularPosition = angularPosition;
            Speed = speed;
            AngularSpeed = angularSpeed;
        }

        #endregion

        #region Propriétés et accesseurs

        /// <summary>
        /// Coordonnée monde du centre de masse de l'objet selon l'axe X
        /// </summary>
        public float X
        {
            get
            {
                return position[0, 0];
            }
            set
            {
                position[0, 0] = value;
            }
        }

        /// <summary>
        /// Coordonnée monde du centre de masse de l'objet selon l'axe Y
        /// </summary>
        public float Y
        {
            get
            {
                return position[1, 0];
            }
            set
            {
                position[1, 0] = value;
            }
        }

        /// <summary>
        /// Coordonnée monde du centre de masse de l'objet selon l'axe Z
        /// </summary>
        public float Z
        {
            get
            {
                return position[2, 0];
            }
            set
            {
                position[2, 0] = value;
            }
        }

        /// <summary>
        /// Position du centre de masse de l'objet en coordonnées monde
        /// </summary>
        public Mapack.Matrix Position
        {
            get
            {
                return position.Clone();
            }
            set
```

```
         {
            position[0, 0] = value[0, 0];
            position[1, 0] = value[1, 0];
            position[2, 0] = value[2, 0];
         }
      }

      /// <summary>
      /// Quaternion représentant l'orientation de l'objet
      /// </summary>
      public Quaternion AngularPosition
      {
         get
         {
            return angularPosition;
         }
         set
         {
            angularPosition.X = value.X;
            angularPosition.Y = value.Y;
            angularPosition.Z = value.Z;
            angularPosition.W = value.W;
         }
      }

      /// <summary>
      /// Vecteur vitesse de l'objet
      /// </summary>
      public Mapack.Matrix Speed
      {
         get
         {
            return speed.Clone();
         }
         set
         {
            speed[0, 0] = value[0, 0];
            speed[1, 0] = value[1, 0];
            speed[2, 0] = value[2, 0];
         }
      }

      /// <summary>
      /// Vecteur vitesse de rotation de l'objet
      /// </summary>
      public Mapack.Matrix AngularSpeed
      {
         get
         {
            return angularSpeed.Clone();
         }
         set
         {
            angularSpeed[0, 0] = value[0, 0];
            angularSpeed[1, 0] = value[1, 0];
            angularSpeed[2, 0] = value[2, 0];
         }
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Calcule la vitesse d'une point du solide
      /// </summary>
```

```
      /// <param name="point">Point dont il faut calculer la vitesse</param>
      /// <returns>Vitesse du point</returns>
      public virtual Mapack.Matrix PointSpeed(Mapack.Matrix point)
      {
         return (Speed + Vector3.Cross(Position − point, AngularSpeed));
      }

      #endregion
   }
}
```

```
using System;
using System.Collections;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe abstraite qui définit et implémente les caractéristiques physiques
 de base d'un solide.
   /// </summary>
   public abstract class KinemaBaseObject
   {
      #region Variables de la classe

      protected KinemaEngine engine;
      protected DynamicState state, nextState = new DynamicState();
      protected Vector3 forces = Vector3.Empty, torques = Vector3.Empty;
      protected Matrix worldMatrix = Matrix.Identity, nextWorldMatrix = Matrix.I
dentity;
      protected Mapack.Matrix inertialTensor = new Mapack.Matrix(3, 3, 1.0);
      protected Mapack.Matrix inversedInertialTensor = new Mapack.Matrix(3, 3, 1
.0);
      protected float scaling = 1.0f, mass = 1.0f;
      protected BoundingSphereTree boundingSphereTree = new BoundingSphereTree()
;
      protected int boundingSphereTreeLevelDrawing = 0;
      protected bool isWorldMatrixCalculated = false, isNextWorldMatrixCalculate
d = false, isDynamic = true, isCollidable = true;
      protected long iD;

      // Variable globale à tous les membres de la classe utilisée pour la génér
ation des ID d'objets
      protected static long numberOfObjectCreated = 0;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur.
      /// ATTENTION : CE CONSTRUCTEUR DOIT ETRE APPELLE PAR TOUS LES OBJETS
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel l'objet appartient</param>
      /// <param name="position">Position initiale du centre de masse</param>
      /// <param name="angularPosition">Position angulaire initiale, représentée
 par un quaternion</param>
      /// <param name="speed">Vitesse initiale du centre de masse</param>
      /// <param name="rotSpeed">Vitesse angulaire initiale</param>
      public KinemaBaseObject(KinemaEngine kinemaEngine, Vector3 position,
         Quaternion angularPosition, Vector3 speed, Vector3 angularSpeed)
      {
         engine = kinemaEngine;
         State = new DynamicState(position, angularPosition, speed, angularSpeed
);
         ID = numberOfObjectCreated;
         numberOfObjectCreated++;
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Identifiant unique pour chaque objet
```

```
      /// </summary>
      public long ID
      {
         get
         {
            return iD;
         }
         private set
         {
            iD = value;
         }
      }

      /// <summary>
      /// Indique si le solide est dynamique
      /// <see langword="false"/> signifie que le solide sera considéré comme fi
xe lors de la résolution dynamique
      /// </summary>
      public bool IsDynamic
      {
         get
         {
            return isDynamic;
         }
         set
         {
            isDynamic = value;
         }
      }

      /// <summary>
      /// Indique si le solide est soumis aux règles de collisions
      /// </summary>
      public bool IsCollidable
      {
         get
         {
            return isCollidable;
         }
         set
         {
            isCollidable = value;
         }
      }

      /// <summary>
      /// Indique si les contraintes sur l'objet sont valides ou non
      /// </summary>
      public virtual bool IsValid
      {
         get
         {
            return true;
         }
      }

      /// <summary>
      /// Niveau de l'arbre à dessiner
      /// 0 : aucun dessin
      /// −1 : dessine toutes les feuilles
      /// </summary>
      public int BoundingSphereTreeLevelDrawing
      {
         get
         {
            return boundingSphereTreeLevelDrawing;
```

```
        }
        set
        {
            boundingSphereTreeLevelDrawing = value;
        }
    }

    /// <summary>
    /// Valeur de mise à l'échelle du modèle de l'objet
    /// </summary>
    public float Scaling
    {
        get
        {
            return scaling;
        }
        set
        {
            scaling = value;
        }
    }

    /// <summary>
    /// Paramètres cinématiques de l'objet
    /// </summary>
    public DynamicState State
    {
        get
        {
            return state;
        }
        set
        {
            state = new DynamicState(value);
            isWorldMatrixCalculated = false;
        }
    }

    /// <summary>
    /// Paramètres cinématiques de l'objet après mise à jour
    /// </summary>
    public DynamicState NextState
    {
        get
        {
            return nextState;
        }
        set
        {
            nextState = new DynamicState(value);
            isNextWorldMatrixCalculated = false;
        }
    }

    /// <summary>
    /// Résultante des forces extérieures appliquées à l'objet
    /// </summary>
    public Vector3 Forces
    {
        get
        {
            return forces;
        }
        set
        {
            forces.X = value.X;
```

```
            forces.Y = value.Y;
            forces.Z = value.Z;
         }
      }

      /// <summary>
      /// Résultante des couples extérieurs appliqués à l'objets,
      /// exprimé au centre de masse de l'objet
      /// </summary>
      public Vector3 Torques
      {
         get
         {
            return torques;
         }
         set
         {
            torques.X = value.X;
            torques.Y = value.Y;
            torques.Z = value.Z;
         }
      }

      /// <summary>
      /// Masse de l'objet
      /// </summary>
      public float Mass
      {
         get
         {
            return mass;
         }
         set
         {
            mass = value;
         }
      }

      /// <summary>
      /// Matrice d'inertie de l'objet, exprimée au centre de masse de l'objet
      /// </summary>
      public Mapack.Matrix InertialTensor
      {
         get
         {
            return inertialTensor;
         }
         set
         {
            inertialTensor = value.Clone();
            inversedInertialTensor = inertialTensor.Inverse;
         }
      }

      /// <summary>
      /// Inverse de la matrice d'inertie de l'objet exprimée au centre de masse
de l'objet
      /// </summary>
      protected Mapack.Matrix InversedInertialTensor
      {
         get
         {
            return inversedInertialTensor;
         }
      }
```

```
      /// <summary>
      /// Matrice de transformation monde de l'objet
      /// </summary>
      public Matrix WorldMatrix
      {
         get
         {
            if (!isWorldMatrixCalculated)
            {
               worldMatrix = CalculateWorldMatrix(State);
               isWorldMatrixCalculated = true;
            }
            return worldMatrix;
         }
         protected set
         {
            worldMatrix = value;
         }
      }

      /// <summary>
      /// Matrice de transformation monde de l'objet dans son état après mise à
jour
      /// </summary>
      public Matrix NextWorldMatrix
      {
         get
         {
            if (!isNextWorldMatrixCalculated)
            {
               nextWorldMatrix = CalculateWorldMatrix(NextState);
               isNextWorldMatrixCalculated = true;
            }
            return nextWorldMatrix;
         }
         protected set
         {
            nextWorldMatrix = value;
         }
      }

      /// <summary>
      /// Met à jour la matrice de transformation de l'arbre de collision
      /// </summary>
      /// <returns>Arbre de collision mis à jour</returns>
      public virtual BoundingSphereTree GetBoundingSphereTree()
      {
         boundingSphereTree.UpdateWorldMatrix(WorldMatrix, Scaling);
         return boundingSphereTree;
      }

      #endregion

      #region Gestion du device

      /// <summary>
      /// Gère le reset du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public abstract void OnDeviceReset(object sender, EventArgs e);

      /// <summary>
      /// Gère la perte du device
      /// </summary>
      /// <param name="sender">sender</param>
```

```
        /// <param name="e">event</param>
        public abstract void OnDeviceLost(object sender, EventArgs e);

        /// <summary>
        /// Gère la libération des ressources à la destruction du device
        /// </summary>
        /// <param name="sender">sender</param>
        /// <param name="e">event</param>
        public abstract void OnDeviceDisposing(object sender, EventArgs e);

        #endregion

        #region Opérations de mise à jour

        /// <summary>
        /// Met à jour l'objet et lui applique de PFD
        /// La mise à jour est écrite dans l'état NextState
        /// </summary>
        /// <param name="deltatime">Durée du pas de calcul de la frame</param>
        public virtual void Update(double deltatime)
        {
            // On calcule la nouvelle vitesse et la nouvelle position par intégrati
on du PFD
            NextState.Position = State.Position + (float)deltatime * State.Speed +
(float)((deltatime * deltatime / (2 * Mass)) * Forces;
            NextState.Speed = State.Speed + (float)(deltatime / Mass) * Forces;

            // On détermine la nouvelle orientation an utilisant la vitesse de rota
tion à t
            NextState.AngularPosition = State.AngularPosition *
               Quaternion.RotationAxis(Vector3.Normalize(State.AngularSpeed), (floa
t)deltatime * State.AngularSpeed.Length());
            //AngularPosition.Normalize();

            // On détermine la nouvelle vitesse de rotation grâce au PFD
            // La matrice d'inertie étant exprimée en coordonnées objet, on transfo
rme le couple extérieur
            // et la vitesse de rotation en coordonnées objets, on obtient alors l'
accélération en coordonnées objets
            // que l'on transforme en coordonnées monde

            // Matrices de transformation des vecteurs (pas de mise à l'échelle, ni
 de translation)
            Matrix vectorTransformObjectToWorld = Matrix.RotationQuaternion(State.A
ngularPosition);
            Matrix vectorTransformWorldToObject = Matrix.Invert(vectorTransformObje
ctToWorld);
            // Vecteur rotation en coordonnées objet
            Vector3 rotSpeObjectCoord = Vector3.TransformNormal(State.AngularSpeed,
 vectorTransformWorldToObject);
            Mapack.Matrix rotationSpeedObjectCoord = MatrixHelper.MatrixFromVector3
(rotSpeObjectCoord);
            // Couple extérieur en coordonnées objet
            Vector3 torquesObjectCoord = Vector3.TransformNormal(Torques, vectorTra
nsformWorldToObject);

            // Accélération en coordonnées objet
            Mapack.Matrix rotationAcceleration;
            rotationAcceleration = InversedInertialTensor * MatrixHelper.MatrixFrom
Vector3( torquesObjectCoord −
               Vector3.Cross(rotSpeObjectCoord, MatrixHelper.Vector3FromMatrix(Iner
tialTensor * rotationSpeedObjectCoord)));

            // On retransforme l'accélération en coordonnées monde avant de l'appli
quer
            Vector3 rotAcc = MatrixHelper.Vector3FromMatrix(rotationAcceleration);
```

```
         rotAcc.TransformNormal(vectorTransformObjectToWorld);
         NextState.AngularSpeed = State.AngularSpeed + (float)deltatime * rotAcc
;

         isNextWorldMatrixCalculated = false;
      }

      /// <summary>
      /// Ecrit l'état mis à jour dans l'état courant
      /// </summary>
      public virtual void FinalizeUpdate()
      {
         State = NextState;

         isWorldMatrixCalculated = false;
      }

      /// <summary>
      /// Dessine l'objet sur le device
      /// </summary>
      public abstract void Render();

      /// <summary>
      /// Calcule la matrice de transformation monde de l'objet
      /// </summary>
      protected virtual Matrix CalculateWorldMatrix(DynamicState state)
      {
         // La matrice consiste en, dans cet ordre :
         // - Mise à l'échelle du modèle selon scaling
         // - Rotation du modèle selon le quaternion angularPosition
         // - Translation du modèle selon le vecteur position

         Matrix world = Matrix.Identity;
         world.AffineTransformation(Scaling, Vector3.Empty, state.AngularPositio
n, state.Position);

         return world;
      }

      #endregion

      #region Méthodes d'application d'une impulsion/d'un moment angulaire

      /// <summary>
      /// Applique une impulsion à l'objet dans son état courant
      /// </summary>
      /// <param name="p">Impulsion à appliquer</param>
      /// <param name="point">Point d'application en coordonnées monde</param>
      public void ApplyImpulseAtPoint(Vector3 p, Vector3 point)
      {
         // Si le solide n'est pas dynamique, il ne réagit pas
         if (!IsDynamic)
            return;

         // Sinon on calcule la variation de vitesse angulaire
         Vector3 deltaAS = MatrixHelper.Vector3FromMatrix(DynamicSolver.WiP(this
, point) * MatrixHelper.MatrixFromVector3(p));

         State.AngularSpeed += deltaAS;

         // Puis la variation de vitesse du centre d'inertie
         State.Speed += 1 / Mass * p;
      }

      /// <summary>
      /// Applique un moment angulaire au solide dans son état courant
```

```
      /// </summary>
      /// <param name="l">Moment angulaire à appliquer</param>
      public void ApplyAngularMomentum(Vector3 l)
      {
         // Si le solide n'est pas dynamique, il ne réagit pas
         if (!IsDynamic)
            return;

         // Sinon on calcule la variation de vitesse angulaire
         Vector3 deltaAS = MatrixHelper.Vector3FromMatrix(DynamicSolver.Li(this)
 * MatrixHelper.MatrixFromVector3(l));

         State.AngularSpeed += deltaAS;
      }

      #endregion

      #region Méthodes helper

      /// <summary>
      /// Calcule la matrice d'inertie en coordonnées monde pour l'état courant
de l'objet
      /// </summary>
      /// <returns>Matrice d'inertie en coordonnées monde pour l'état courant de
 l'objet</returns>
      public Mapack.Matrix GetInertialTensorInStatesWorldCoord()
      {
         Matrix world = WorldMatrix;
         Mapack.Matrix transform = MatrixHelper.DirectXToMapackMatrix33(world).T
ranspose();
         world.Invert();
         Mapack.Matrix invertTransform = MatrixHelper.DirectXToMapackMatrix33(wo
rld).Transpose();
         Mapack.Matrix result = Mapack.Matrix.Multiply(transform, InertialTensor
);
         result = Mapack.Matrix.Multiply(result, invertTransform);

         return result;
      }

      /// <summary>
      /// Calcule la matrice d'inertie en coordonnées monde pour l'état après mi
se à jour de l'objet
      /// </summary>
      /// <returns>Matrice d'inertie en coordonnées monde pour l'état après mise
 à jour de l'objet</returns>
      public Mapack.Matrix GetInertialTensorInNextStatesWorldCoord()
      {
         Matrix world = NextWorldMatrix;
         Mapack.Matrix transform = MatrixHelper.DirectXToMapackMatrix33(world).T
ranspose();
         world.Invert();
         Mapack.Matrix invertTransform = MatrixHelper.DirectXToMapackMatrix33(wo
rld).Transpose();
         Mapack.Matrix result = Mapack.Matrix.Multiply(transform, InertialTensor
);
         result = Mapack.Matrix.Multiply(result, invertTransform);

         return result;
      }

      /// <summary>
      /// Détermine si 2 objets sont en collision.
      /// Récupère les paramètres de collision correspondant aux spheres dont le
s centres sont les plus proches
      /// </summary>
```

```
      /// <param name="object1">Premier objet à tester</param>
      /// <param name="object2">Second objet à tester</param>
      /// <param name="normal">Renvoie la normale normale au contact</param>
      /// <returns>Renvoie <see langword="true"/> si une collision est trouvée</
returns>
      public static bool IsColliding(KinemaBaseObject object1, KinemaBaseObject
object2, out Vector3 normal)
      {
         // On vérifie que les 2 objets obéissent aux lois de collisions
         if (!object1.IsCollidable || !object2.IsCollidable)
         {
            normal = Vector3.Empty;
            return false;
         }

         ArrayList intersectedPairsFrom1, intersectedPairsFrom2;
         normal = Vector3.Empty;

         // On met à jour les positions des arbres
         object1.GetBoundingSphereTree().UpdateWorldMatrix(object1.WorldMatrix,
object1.Scaling);
         object2.GetBoundingSphereTree().UpdateWorldMatrix(object2.WorldMatrix,
object2.Scaling);

         // On teste les collisions entre les arbres
         if (BoundingSphereTree.Intersect(object1.GetBoundingSphereTree(), objec
t2.GetBoundingSphereTree(),
            out intersectedPairsFrom1, out intersectedPairsFrom2) == 0)
         {
            // il n'y a pas de collision
            return false;
         }

         // On détermine parmi les spheres en collisions celles dont les centres
 sont les plus proches
         // sachant que les spheres en question on été transformées pendant leur
 recherche
         int indexOfClosestPair = 0;
         float currentSquaredDistance, squaredDistanceOfClosestPair = Vector3.Le
ngthSq(
            ((BoundingSphereTreeNode)intersectedPairsFrom1[0]).TransformedCenter
            - ((BoundingSphereTreeNode)intersectedPairsFrom2[0]).TransformedCent
er);

         for (int i = 1; i < intersectedPairsFrom1.Count; i++)
         {
            currentSquaredDistance = Vector3.LengthSq(
            ((BoundingSphereTreeNode)intersectedPairsFrom1[i]).TransformedCenter
            - ((BoundingSphereTreeNode)intersectedPairsFrom2[i]).TransformedCent
er);

            if (currentSquaredDistance < squaredDistanceOfClosestPair)
            {
               indexOfClosestPair = i;
               squaredDistanceOfClosestPair = currentSquaredDistance;
            }
         }

         // On détermine la normale au contact et on la normalise
         normal = Vector3.Normalize(((BoundingSphereTreeNode)intersectedPairsFro
m1[indexOfClosestPair]).TransformedCenter
            - ((BoundingSphereTreeNode)intersectedPairsFrom2[indexOfClosestPair]
).TransformedCenter);

         return true;
      }
```

```
      // Méthodes permettant de définir la matrice d'inertie à partir d'une form
e de base
      /// <summary>
      /// Calcule la matrice d'inertie d'une boule
      /// </summary>
      /// <param name="radius">Rayon de la boule</param>
      public void SetSphereInertialTensor(double radius)
      {
         InertialTensor = new Mapack.Matrix(3, 3, 2.0 / 5.0 * Mass * radius * ra
dius);
      }

      /// <summary>
      /// Calcule la matrice d'inertie d'un parallélépipède aligné sur le repère
 objet
      /// </summary>
      /// <param name="X">Taille selon l'axe X</param>
      /// <param name="Y">Taille selon l'axe Y</param>
      /// <param name="Z">Taille selon l'axe Z</param>
      public void SetBoxInertialTensor(double X, double Y, double Z)
      {
         Mapack.Matrix tensor = new Mapack.Matrix(3, 3);
         tensor[0, 0] = Mass / 12.0 * (Y * Y + Z * Z);
         tensor[1, 1] = Mass / 12.0 * (X * X + Z * Z);
         tensor[2, 2] = Mass / 12.0 * (X * X + Y * Y);

         InertialTensor = tensor;
      }

      /// <summary>
      /// Calcule la vitesse d'une point du solide
      /// </summary>
      /// <param name="point">Point dont il faut calculer la vitesse</param>
      /// <returns>Vitesse du point</returns>
      public virtual Vector3 PointSpeed(Vector3 point)
      {
         return State.PointSpeed(point);
      }

      /// <summary>
      /// Normalise une angle
      /// </summary>
      /// <param name="angle">Angle à normaliser</param>
      /// <returns>Valeur de l'angle ramenée dans l'intervalle [0,2.Pi[</returns
>
      protected float NormalizeAngle(float angle)
      {
         while (angle >= 2.0f * (float)Math.PI)
            angle -= 2.0f * (float)Math.PI;

         while (angle < 0.0f)
            angle += 2.0f * (float)Math.PI;

         return angle;
      }

      #endregion

      #region Détermination de l'égalité de 2 objets

      public override bool  Equals(object obj)
      {
         if (!(obj is KinemaBaseObject))
            return false;
         KinemaBaseObject other = (KinemaBaseObject)obj;
```

```
         if (ID == other.ID)
            return true;
         return false;
      }

      public override int GetHashCode()
      {
         return ID.GetHashCode();
      }

      public static bool operator ==(KinemaBaseObject op1, KinemaBaseObject op2)
      {
         return op1.Equals(op2);
      }

      public static bool operator !=(KinemaBaseObject op1, KinemaBaseObject op2)
      {
         return !op1.Equals(op2);
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;

namespace Kinema
{
   /// <summary>
   /// Classe implémentant la représentation de l'objet par un Mesh
   /// </summary>
   public class KinemaMeshObject : KinemaBaseObject
   {
      #region Variables de la classe

      protected Mesh mesh;
      protected Texture[] meshTextures;
      protected Material[] meshMaterials;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur utilisant des ressources déjà chargées.
      /// Les ressources doivent être placées dans le pool Managed.
      /// Attention, le constructeur ne génère pas l'arbre de spheres englobante
s !
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel l'objet appartient</param>
      /// <param name="pos">Position initiale du centre de masse</param>
      /// <param name="angularPos">Position angulaire initiale</param>
      /// <param name="spe">Vitesse initiale du centre de masse</param>
      /// <param name="rotSpe">Vitesse angulaire initiale, représentée par un qu
aternion</param>
      /// <param name="model">Mesh contenant la géométrie de l'objet</param>
      /// <param name="textures">Textures du Mesh</param>
      /// <param name="materials">Matériaux du Mesh</param>
      public KinemaMeshObject(KinemaEngine kinemaEngine, Vector3 pos, Quaternion
 angularPos, Vector3 spe, Vector3 rotSpe,
         Mesh model, Texture[] textures, Material[] materials)
         : base(kinemaEngine, pos, angularPos, spe, rotSpe)

      {
         mesh = model;
         meshTextures = textures;
         meshMaterials = materials;
      }

      /// <summary>
      /// Constructeur chargeant un Mesh depuis un fichier .x vers le pool manag
ed
      /// Attention, le constructeur ne génère pas l'arbre de spheres englobante
s !
      /// </summary>
      /// <param name="kinemaEngine">Moteur auquel l'objet appartient</param>
      /// <param name="pos">Position initiale du centre de masse</param>
      /// <param name="angularPos">Position angulaire initiale, représentée par
un quaternion</param>
      /// <param name="spe">Vitesse initiale du centre de masse</param>
      /// <param name="rotSpe">Vitesse angulaire initiale</param>
      /// <param name="meshFileName">Nom du fichier .x à charger</param>
      public KinemaMeshObject(KinemaEngine kinemaEngine, Vector3 pos, Quaternion
 angularPos, Vector3 spe, Vector3 rotSpe,
         string meshFileName)
         : base(kinemaEngine, pos, angularPos, spe, rotSpe)
```

```
      {
         MeshFromFile(meshFileName);
      }

      #endregion

      #region Gestion du device

      /// <summary>
      /// Gère le reset du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceReset(object sender, EventArgs e)
      {
      }

      /// <summary>
      /// Gère la perte du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceLost(object sender, EventArgs e)
      {
      }

      /// <summary>
      /// Gère la libération des ressources à la destruction du device
      /// </summary>
      /// <param name="sender">sender</param>
      /// <param name="e">event</param>
      public override void OnDeviceDisposing(object sender, EventArgs e)
      {
      }

      #endregion

      #region Opérations de mise à jour

      /// <summary>
      /// Dessine le Mesh sur le device
      /// </summary>
      public override void Render()
      {
         engine.Device.Transform.World = WorldMatrix;

         for (int i = 0; i < meshMaterials.Length; i++)
         {
            // On définit le material et la texture pour ce morceau
            engine.Device.Material = meshMaterials[i];
            engine.Device.SetTexture(0, meshTextures[i]);

            // On dessine le morceau
            mesh.DrawSubset(i);
         }

         // Si demandé, on dessine un des niveaux de l'arbre de spheres engloban
tes
         boundingSphereTree.UpdateWorldMatrix(WorldMatrix, Scaling);
         boundingSphereTree.Render(engine.Device, BoundingSphereTreeLevelDrawing
);
      }

      #endregion

      #region Méthodes helper
```

```
      /// <summary>
      /// Charge un Mesh à partir d'un ficher .x vers le pool Managed
      /// Attention, la méthode ne génère pas l'arbre de spheres englobantes !
      /// </summary>
      /// <param name="fileName">Nom du fichier .x à charger</param>
      public void MeshFromFile(string fileName)
      {
         ExtendedMaterial[] materials = null;

         mesh = Mesh.FromFile(fileName, MeshFlags.Managed, engine.Device, out ma
terials);

         // On récuprère les propriétés des materials et les noms des textures
         meshTextures = new Texture[materials.Length];
         meshMaterials = new Material[materials.Length];
         for (int i = 0; i < materials.Length; i++)
         {
            meshMaterials[i] = materials[i].Material3D;

            // On définit l'"ambient color" du material
            meshMaterials[i].Ambient = meshMaterials[i].Diffuse;

            // On crée la texture
            meshTextures[i] = TextureLoader.FromFile(engine.Device, materials[i]
.TextureFilename);
         }
      }

      #endregion

      #region Génération des arbres de spheres englobantes

      /// <summary>
      /// Génère un arbre binaire à partir du Mesh chargé
      /// </summary>
      public void GenerateBinaryTree()
      {
         if (mesh == null)
            throw new Exception("Pas de Mesh chargé, impossible de générer l'arb
re de spheres englobantes");

         boundingSphereTree.GenerateBinaryTreeFromMesh(mesh);
      }

      /// <summary>
      /// Génère un octree de spheres à partir du Mesh chargé
      /// </summary>
      /// <param name="level">Niveau de division voulu dans l'arbre</param>
      /// <param name="samplePointsPrecision">Precision de la génération du nuag
e de points représentant l'objet</param>
      public void GenerateOctree(int level, int samplePointsPrecision)
      {
         if (mesh == null)
            throw new Exception("Pas de Mesh chargé, impossible de générer l'arb
re de spheres englobantes");

         boundingSphereTree.GenerateOctreeFromMesh(mesh, level, samplePointsPrec
ision);
      }

      #endregion
   }
}
```

```
using System;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.DirectInput;

namespace Kinema
{
   /// <summary>
   /// Classe permettant de gérer les entrées clavier de base
   /// </summary>
   public class Keyboard
   {
      #region Variables de la classe

      protected Device device;
      protected KeyboardState state;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur
      /// </summary>
      /// <param name="form">Fenêtre accueillant l'application</param>
      public Keyboard(Form form)
      {
         // On crée le device qui gère le clavier
         device = new Device(SystemGuid.Keyboard);
         device.SetCooperativeLevel(form, CooperativeLevelFlags.Background | Coo
perativeLevelFlags.NonExclusive);
         device.SetDataFormat(DeviceDataFormat.Keyboard);

         try
         {
            device.Acquire();
         }
         catch (DirectXException ex)
         {
            Console.WriteLine(ex.Message);
         }
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Renvoie l'état actuel du clavier
      /// </summary>
      public KeyboardState State
      {
         get
         {
            return state;
         }
         protected set
         {
            state = value;
         }
      }

      /// <summary>
      /// Indexer qui permet d'obtenir directement l'état d'une touche
      /// </summary>
      /// <param name="index">touche à tester, la fonction renvoie false si l'in
```

```
dex est (<see langword="Key"/>)(-1)</param>
      /// <returns><see langword="true"/> si la touche est enfoncée</returns>
      public bool this[Key index]
      {
         get
         {
            if (index == (Key)(-1))
               return false;

            return State[index];
         }
      }

      #endregion

      #region Communication avec le device

      /// <summary>
      /// Communique avec le device pour mettre à jour l'état du clavier
      /// </summary>
      public void Poll()
      {
         // On essaie de récupérer les informations du clavier
         try
         {
            device.Poll();
            State = device.GetCurrentKeyboardState();
         }
         catch (NotAcquiredException)
         {
            // En cas de problème, on essaie de récupérer le device du clavier
            try
            {
               device.Acquire();
            }
            catch (InputException iex)
            {
               Console.WriteLine(iex.Message);
               // Impossible de récupérer le device du clavier
            }
         }
         catch (InputException ex2)
         {
            Console.WriteLine(ex2.Message);
         }
      }

      #endregion

      #region Gestion des ressources

      /// <summary>
      /// Libère les ressources utilisées par la classe
      /// </summary>
      public void Dispose()
      {
         if (device != null)
         {
            device.Unacquire();
            device.Dispose();
         }
      }

      #endregion
   }
}
```

```
using System;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.DirectInput;

namespace Kinema
{
   /// <summary>
   /// Classe permettant de gérer les entrées souris de base
   /// </summary>
   public class Mouse
   {
      #region Variables de la classe

      protected Device device;
      protected bool axisModeAbsolute;
      protected MouseState state;
      protected byte[] buttonBuffer;
      protected int dx, dy, dz;
      protected double vx = 0, vy = 0, vz = 0;

      #endregion

      #region Constructeur(s)

      /// <summary>
      /// Constructeur
      /// </summary>
      /// <param name="form">Fenêtre accueillant l'application</param>
      /// <param name="axisModeAbsolute">Si mis à <see langword="true"/>, la cla
sse lit les coordonnées écran du pointeur</param>
      public Mouse(Form form, bool axisModeAbsolute)
      {
         AxisModeAbsolute = axisModeAbsolute;

         // On crée le device qui gère la souris
         device = new Device(SystemGuid.Mouse);
         device.SetCooperativeLevel(form, CooperativeLevelFlags.Background | Coo
perativeLevelFlags.NonExclusive);
         device.SetDataFormat(DeviceDataFormat.Mouse);

         try
         {
            device.Acquire();
            device.Properties.AxisModeAbsolute = AxisModeAbsolute;
         }
         catch (DirectXException dex)
         {
            Console.WriteLine(dex.Message);
         }
      }

      #endregion

      #region Propriétés et accesseurs

      /// <summary>
      /// Si à <see langword="true"/>, la classe lit les coordonnées écran du po
inteur
      /// Si à <see langword="false"/>, la classe lit les déplacements relatifs
du pointeur
      /// </summary>
      public bool AxisModeAbsolute
      {
         get
         {
```

```
            return axisModeAbsolute;
         }
         set
         {
            axisModeAbsolute = value;
         }
      }

      /// <summary>
      /// Dernier déplacement du pointeur selon l'axe x
      /// </summary>
      public int Dx
      {
         get
         {
            return dx;
         }
         protected set
         {
            dx = value;
         }
      }
      /// <summary>
      /// Dernier déplacement du pointeur selon l'axe y
      /// </summary>
      public int Dy
      {
         get
         {
            return dy;
         }
         protected set
         {
            dy = value;
         }
      }
      /// <summary>
      /// Dernier déplacement de la molette
      /// </summary>
      public int Dz
      {
         get
         {
            return dz;
         }
         protected set
         {
            dz = value;
         }
      }

      /// <summary>
      /// Vitesse de déplacement du pointeur selon l'axe x, en pixels par second
e
      /// </summary>
      public double XSpeed
      {
         get
         {
            return vx;
         }
         protected set
         {
            vx = value;
         }
      }
```

```
        /// <summary>
        /// Vitesse de déplacement du pointeur selon l'axe y, en pixels par second
e
        /// </summary>
        public double YSpeed
        {
            get
            {
                return vy;
            }
            protected set
            {
                vy = value;
            }
        }
        /// <summary>
        /// Vitesse de déplacement de la molette, en unités par seconde
        /// </summary>
        public double ZSpeed
        {
            get
            {
                return vz;
            }
            protected set
            {
                vz = value;
            }
        }

        /// <summary>
        /// Renvoie l'état de la souris
        /// </summary>
        public MouseState State
        {
            get
            {
                return state;
            }
            protected set
            {
                state = value;
            }
        }

        /// <summary>
        /// Renvoie l'état des boutons de la souris
        /// </summary>
        public byte[] MouseButtons
        {
            get
            {
                return buttonBuffer;
            }
            protected set
            {
                buttonBuffer = value;
            }
        }

        #endregion

        #region Communication avec le device

        /// <summary>
        /// Communique avec le device pour mettre à jour l'état du clavier
```

```
      /// Surcharge par défaut qui ne calcule pas les vitesses de déplacement
      /// </summary>
      public void Poll()
      {
         Poll(0);
      }

      /// <summary>
      /// Communique avec le device pour mettre à jour l'état du clavier
      /// </summary>
      /// <param name="deltatime">
      /// Durée du pas de calcul de la frame, utilisé pour le calcul des vitesse
s si non nul</param>
      public void Poll(double deltatime)
      {
         // On essaie de récupérer l'état de la souris
         try
         {
            MouseState previousState = State;
            device.Poll();
            State = device.CurrentMouseState;
            MouseButtons = State.GetMouseButtons();

            // On calcule le déplacement de la souris
            if (!AxisModeAbsolute)
            {
               // State donne direcement le déplacement
               Dx = State.X;
               Dy = State.Y;
               Dz = State.Z;
            }
            else
            {
               // State donne la position écran de la souris
               Dx = State.X - previousState.X;
               Dy = State.Y - previousState.Y;
               Dz = State.Z - previousState.Z;
            }

            // On calcule la vitesse de la souris si demandé
            if (deltatime != 0)
            {
               XSpeed = Dx / deltatime;
               YSpeed = Dy / deltatime;
               ZSpeed = Dz / deltatime;
            }
         }
         catch (NotAcquiredException)
         {
            // En cas de problème, on essaie de récupérer le device de la souris
            try
            {
               device.Acquire();
            }
            catch (InputException iex)
            {
               Console.WriteLine(iex.Message);
               // impossible de récupérer le device de la souris
            }
         }
         catch (InputException ex2)
         {
            Console.WriteLine(ex2.Message);
         }
      }
```

```
      #endregion

      #region Gestion des ressources

      /// <summary>
      /// Libère les ressources utilisées par la classe
      /// </summary>
      public void Dispose()
      {
         device.Unacquire();
         device.Dispose();
      }

      #endregion
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Kinema
{
   /// <summary>
   /// Classe assurant le calcul du framerate
   /// </summary>
   class FrameRateHelper
   {
      /// <summary>
      /// Calcule le framerate
      /// </summary>
      /// <returns>framerate calculé</returns>
      public static int CalculateFrameRate ( )
      {
         if ( System.Environment.TickCount - lastTick >= 1000 )
         {
            lastFrameRate = frameRate;
            frameRate = 0;
            lastTick = System.Environment.TickCount;
         }

         frameRate++;

         return lastFrameRate;
      }

      private static int lastTick;
      private static int lastFrameRate;
      private static int frameRate;
    }
}
```

```
using System;
using System.Runtime.InteropServices;

namespace Kinema
{
    /// <summary>
    /// Class allowing a precise measure of time
    /// </summary>
    class HiResTimer
    {
        private HiResTimer ( ) { } // No creation

        /// <summary>
        /// Static creation routine
        /// </summary>
        static HiResTimer ( )
        {
            isTimerStopped = true;
            ticksPerSecond = 0;
            stopTime = 0;
            lastElapsedTime = 0;
            baseTime = 0;
            // Use QueryPerformanceFrequency to get frequency of the timer
            isUsingQPF = QueryPerformanceFrequency ( ref ticksPerSecond );
        }

        /// <summary>
        /// Resets the timer
        /// </summary>
        public static void Reset ( )
        {
            if ( !isUsingQPF )
                return; // Nothing to do

            // Get either the current time or the stop time
            long time = 0;
            if ( stopTime != 0 )
                time = stopTime;
            else
                QueryPerformanceCounter ( ref time );

            baseTime = time;
            lastElapsedTime = time;
            stopTime = 0;
            isTimerStopped = false;
        }

        /// <summary>
        /// Starts the timer
        /// </summary>
        public static void Start ( )
        {
            if ( !isUsingQPF )
                return; // Nothing to do

            // Get either the current time or the stop time
            long time = 0;
            if ( stopTime != 0 )
                time = stopTime;
            else
                QueryPerformanceCounter ( ref time );

            if ( isTimerStopped )
                baseTime += ( time - stopTime );
            stopTime = 0;
            lastElapsedTime = time;
```

```
            isTimerStopped = false;
        }

        /// <summary>
        /// Stop (or pause) the timer
        /// </summary>
        public static void Stop ( )
        {
            if ( !isUsingQPF )
                return; // Nothing to do

            if ( !isTimerStopped )
            {
                // Get either the current time or the stop time
                long time = 0;
                if ( stopTime != 0 )
                    time = stopTime;
                else
                    QueryPerformanceCounter ( ref time );

                stopTime = time;
                lastElapsedTime = time;
                isTimerStopped = true;
            }
        }

        /// <summary>
        /// Advance the timer a tenth of a second
        /// </summary>
        public static void Advance ( )
        {
            if ( !isUsingQPF )
                return; // Nothing to do

            stopTime += ticksPerSecond / 10;
        }

        /// <summary>
        /// Get the absolute system time
        /// </summary>
        public static double GetAbsoluteTime ( )
        {
            if ( !isUsingQPF )
                return -1.0; // Nothing to do

            // Get either the current time or the stop time
            long time = 0;
            if ( stopTime != 0 )
                time = stopTime;
            else
                QueryPerformanceCounter ( ref time );

            double absolueTime = time / (double)ticksPerSecond;
            return absolueTime;
        }

        /// <summary>
        /// Get the current time
        /// </summary>
        public static double GetTime ( )
        {
            if ( !isUsingQPF )
                return -1.0; // Nothing to do

            // Get either the current time or the stop time
            long time = 0;
```

```
            if ( stopTime != 0 )
                time = stopTime;
            else
                QueryPerformanceCounter ( ref time );

            double appTime = (double)( time − baseTime ) / (double)ticksPerSecon
d;
            return appTime;
        }

        /// <summary>
        /// get the time that elapsed between GetElapsedTime() calls
        /// </summary>
        public static double GetElapsedTime ( )
        {
            if ( !isUsingQPF )
                return −1.0; // Nothing to do

            // Get either the current time or the stop time
            long time = 0;
            if ( stopTime != 0 )
                time = stopTime;
            else
                QueryPerformanceCounter ( ref time );

            double elapsedTime = (double)( time − lastElapsedTime ) / (double)ti
cksPerSecond;
            lastElapsedTime = time;
            return elapsedTime;
        }

        /// <summary>
        /// Returns true if timer stopped
        /// </summary>
        public static bool IsStopped
        {
            get { return isTimerStopped; }
        }

        private static bool isUsingQPF;
        private static bool isTimerStopped;
        private static long ticksPerSecond;
        private static long stopTime;
        private static long lastElapsedTime;
        private static long baseTime;


        [System.Security.SuppressUnmanagedCodeSecurity] // We won’t use this mal
iciously
        [DllImport ( "kernel32" )]
        public static extern bool QueryPerformanceFrequency ( ref long Performan
ceFrequency );

        [System.Security.SuppressUnmanagedCodeSecurity] // We won’t use this mal
iciously
        [DllImport ( "kernel32" )]
        public static extern bool QueryPerformanceCounter ( ref long Performance
Count );
    }
}
```