Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 2961-2974 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

ARQUITECTURA ORIENTADA A COMPONENTES BASADA EN REFLEXIÓN PARA MOTORES FÍSICOS

Pablo S. Rojas Fredini y Alejandro C. Limache

Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC) INTEC-CONICET. Santa Fe, Argentina, srojasfredini@santafe-conicet.gov.ar , http://www.cimec.gov.ar/

Palabras Clave: reflexión, motor físico, patrón de diseño, diseño orientado a componentes

Resumen.

Junto con la evolución de las computadoras, han surgido nuevos lenguajes y paradigmas de programación con características destacables que abren interesantes posibilidades en el campo de la simulación y de la visualización en tiempo real. En el presente trabajo se explotan algunos de estos nuevos conceptos para el desarrollo de un motor de simulación física en tiempo real. El motor de simulación utiliza primariamente una arquitectura orientada a componentes y basada en la propiedad de reflexión que poseen algunos lenguajes modernos como los basados en .NET de Microsoft. El nuevo diseño permite desarrollar simuladores de objetos físicos cuyas propiedades pueden ser modificadas en tiempo de ejecución evitando la necesidad de crear interfaces con lenguajes externos de "scripting". El diseño también permite el agregado de nuevos componentes (con nuevos fenómenos u objetos físicos) y la generación automática de interfaces gráficas y de configuración. Los distintos componentes pueden estar escritos en diferentes lenguajes y pueden agregarse de manera transparente. Como ejemplo concreto de su capacidad, se muestra la aplicación del motor en la simulación simultanea en tiempo real de sólidos rígidos. El objetivo del presente desarrollo es generar una plataforma eficiente para la simulación de vehículos terrestres, aéreos y fluidos en tiempo real.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1. INTRODUCCIÓN

La simulación en tiempo real es un campo de constante innovación e investigación (García Bauza C. , Boroni G. , Vénere M. , Clausse A., 2010; Lazo M. , García Bauza C. y Clausse A., 2009; García Bauza C., Lotito P. , Parente L. , Vénere M., 2008) así como también la metodología para implementar los simuladores y motores físicos (Tasora A. , Anitescu M. , 2009; Deltaknowledge, 2011; Rojas Fredini P.S. , Limache A.C., 2010; Limache A.C. , Rojas Fredini P.S. y Murillo M.H., 2010).

Con la constante evolución del hardware de las computadoras, los lenguajes de programación han ido evolucionando (The C++ Standards Committee, 2011) e incluso han surgido nuevos lenguajes y paradigmas con características importantes. En particular, la tecnología conocida como .NET (Microsoft, 2011) introdujo un framework para aplicaciones que brinda una biblioteca de clases comun que puede ser utilizada desde distintos lenguajes. Este framework funciona dentro de una máquina virtual que se encarga de diversos detalles tales como manejo de memoria, dominios de aplicación, etc. El código que se desarrolla para ejecutar en la plataforma .NET se lo conoce como administrado. Dos de las características más importantes que han sido introducidas son la reflexión que es la capacidad que posee un programa para examinarse y modificarse a sí mismo y la posibilidad de mezclar código nativo con código administrado.

Por otro lado, en los últimos tiempos el gran impulso que tuvo la simulación en tiempo real ha sido dado por la industria de los videojuegos, lugar donde mayor aplicación ha tenido esta disciplina(Unity Technologies, 2011; Epic Games Inc., 2011; Garage Games, 2011). Esto ha ido acompañado por la introducción de nuevos diseños y patrones de software acorde a las necesidades. En particular la utilización de un diseño orientado a componentes es uno de los aspectos más interesantes aportados (Nystrom R., 2010). El mismo es un diseño que compite con la tradicional herencia utilizada en la programación orientada a objetos.

Utilizando las ideas introducidas por los motores de videojuegos y las posibilidades que aportan los nuevos lenguajes se desarrolló un motor de simulación en tiempo real. El mismo está basado en la característica de reflexión y es orientado a componentes. Este motor denominado *Air* es de propósito general, puede ser usado desde diferentes lenguajes y aporta características novedosas en el campo de la simulación. En particular el desarrollo fue motivado por la necesidad de una plataforma de simulación eficiente y flexible para la implementación de simuladores de vehículos terrestres, aéreos y fluidos en tiempo real. Dichos simuladores deben permitir simular varios vehículos de manera concurrente, e incluso a través de redes de comunicación para el entrenamiento simultáneo de pilotos.

2. DISEÑO

En esta sección se hará una introducción a las características más importantes de *Air* y se explicará que ventajas aportan. La implementación del núcleo del simulador se realizó en C#.

2.1. Orientación a componentes

2.1.1. Primer acercamiento

Como se dijo en la introducción Air es un motor orientado a componentes. El patrón de diseño de componentes se basa en la idea de que los objetos del mundo virtual pueden ser descompuestos en distintas partes o componentes, donde cada componente implementa un aspecto del comportamiento del objeto del cual son parte. Esto se contrapone con la tradicional orientación a objetos, dónde un objeto debe heredar de otro para implementar su comportamiento.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Para ser mas precisos los componentes se basan en agregación y no en herencia.

A continuación se presenta un ejemplo. En todo motor de simulación se necesita visualizar los objetos y calcular su comportamiento físico. Entonces se define una clase que realiza la visualización llamada *MeshRenderer*, y otra que realiza los cálculos físicos de cuerpos rígidos llamada *RigidBody*. Con estas clases se desea implementar una nueva clase *Avion* que modele un aeroplano en el mundo simulado. La misma necesita dibujarse y realizar la simulación de cuerpos rígidos. El enfoque tradicional en la orientación a objetos es implementar una herencia como se muestra en la Fig.1.

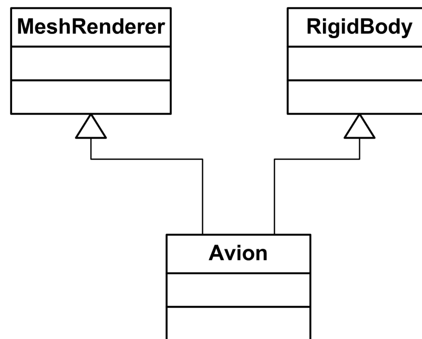

Figura 1: Solución usando herencia

Es decir Avion hereda de MeshRenderer lo que le permite dibujarse mientras que la herencia de RigidBody le permite incorporar el comportamiento de cuerpo rígido. Ahora, si se desea cambiar el comportamiento del simulador para que el avión en vez de comportarse como un cuerpo rígido se empiece a comportar como un cuerpo blando, entonces usando herencia se debería cambiar el RigidBody por la nueva clase SoftBody y volver a compilar el código arreglando todos los cambios introducidos por la nueva herencia, finalmente se obtendría lo que se observa en la Fig.2.

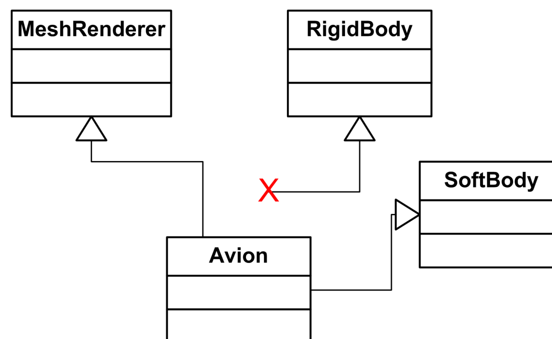

Figura 2: Cambio de herencia

La tarea de cambiar las relaciones de herencia es extensa cuando las jerarquías de objetos son grandes, como en el caso de los simuladores, motores de videojuegos, etc. Esto da como resultado un código poco flexible.

Por otro lado, si se plantea el mismo caso utilizando componentes entonces se definiría un contenedor de componentes denominado Avion el cual contiene una colección de componentes. Dichos componentes le aportan cada uno una parte del comportamiento necesario. Es decir en el ejemplo se tendría un componente de visualización llamado *MeshRendererComponent* y uno

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

de cálculo de cuerpos rígidos llamado *RigidBodyComponent*. Esta nueva jerarquía se puede observar en la Fig.3.

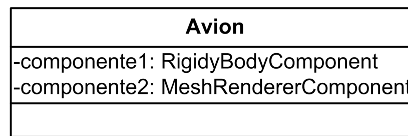

Figura 3: Ejemplo con componentes

Si luego se desea cambiar los cuerpos rígidos por cuerpos blandos, entonces simplemente se debe cambiar el componente *RigidBodyComponente* por uno nuevo el cual llamaremos *SoftBodyComponente*. Este cambio no requirió modificar nada de código y como se mostrará más adelante, ni siquiera volver a compilarlo.

2.1.2. La arquitectura de componentes

En la presente sección se introducirá formalmente la jerarquía de componentes propuesta e implementada en el simulador *Air*. Todo objeto que se instancie dentro de *Air* es un GameObject. Esta clase es una de las más importantes y representa cualquier objeto que pueda existir dentro del mundo virtual. Pueden ser cámaras, vehículos, ventanas de plotting, etc. GameObject por sí misma no posee ningún comportamiento o representación sino que actúa como un contenedor de componentes.

Por otro lado, los componentes son clases que deben heredar de *AirComponent*. *AirComponente* contiene todos los mecanismos necesarios para que el componente se pueda registrar y funcionar correctamente. El diagrama de clases de dicha estructura se puede observar en la Fig. 4

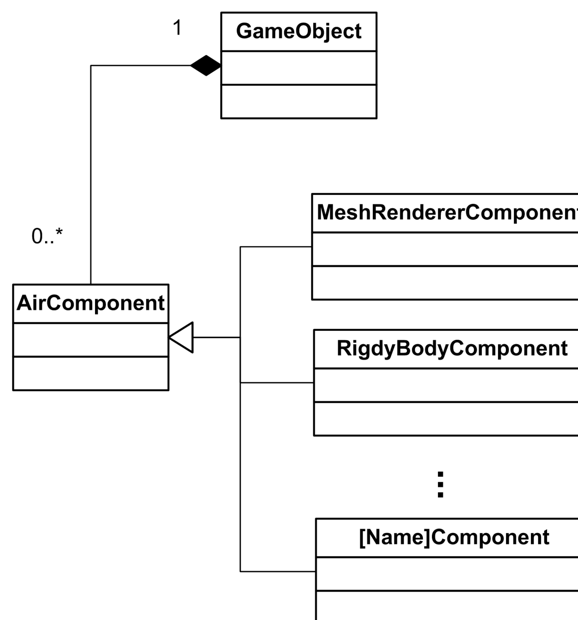

Figura 4: Relación entre GameObject y AirComponent

Los componentes poseen las siguientes características:

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

- Independencia del lenguaje: Un componente puede estar programado en cualquier lenguaje que compile para .NET. Esto da la posibilidad de programar componentes en C++ que interactuen con componentes en Python o C#, etc. Esto permite obtener un correcto equilibrio entre facilidad de desarrollo y potencia.

- Autosuficiencia: Un componente debe ser autosuficiente en el sentido de que el mismo debe poder determinar sus dependencia con otros componentes y solucionarlas en el caso de que alguna este faltando. Ningún componente debe provocar una excepción porque no encuentre una dependencia. Además el componente debe poder navegar por sus componentes hermanos y recuperar la información que necesite para su funcionamiento.

- Creacion y asignación dinámica: Los componentes se pueden agregar o quitar en tiempo de ejecución. Incluso se pueden compilar componentes nuevos sin detener el simulador y enlazarlos dinámicamente.

2.1.3. Eventos de los componentes

Generalmente los componentes necesitan actualizarse en cada paso de tiempo del simulador. Para ello se definieron grupos de actualización. Estos grupos marcan en qué momento se realiza la actualización. Los grupos definidos son:

- Simulación: Los componentes pertenecientes a este grupo reciben tres notificaciones o eventos:

    1. PreUpdate: Es el primero de los eventos y permite inicializar el cálculo de la simulación.
    2. Update: Se llama a continuación del PreUpdate y encapsula la simulación propiamente dicha.
    3. PostUpdate: Es el último evento de simulación que se invoca en un paso de tiempo y permite a los componentes modificarse de acuerdo a los cálculos hechos en el evento anterior.

- Rendering: Los componentes pertencientes a este grupo son aquellos que necesitan ser renderizados. Este grupo es actualizado cuando la API gráfica está lista para dibujar. Este grupo sólo reporta un evento:

    1. Render.

- Pre/post rendering : Este grupo posee dos eventos que se invocan antes y después del evento Render:

    1. PreRender: Utilizado para inicializar la API gráfica y sus dependencias para dibujar el frame o cuadro actual.
    2. PosRender: Permite realizar cualquier tarea adicional una vez que se invoco el método Render de todos los objetos registrados.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

Para que los componentes puedan registrarse como pertenecientes a alguno de los grupos nombrados, los mismos deben implementar la interfaz de grupo correspondiente. Formalmente una interfaz es una definición que contiene sólo las firmas de métodos, delegados o eventos. Toda clase que implemente una interfaz debe proveer la definición de los métodos delegados o eventos que figuran en la misma. En Air, cada grupo posee su interfaz:

- IUpdateable: para el grupo de simulación
- IRenderable: para el grupo de render
- IPrePostRender: para los eventos previos y posteriores al renderizado.

Cada componente puede implementar ninguna o varias de estas interfaces. Es decir, el diagrama de clases de los componentes propuestos en el ejemplo de la sección anterior se puede observar en la Fig. 5. Allí el componente de Render implementa la interfaz IRenderable, mientras que el de simulación de cuerpos rígidos implementa IUpdateable.

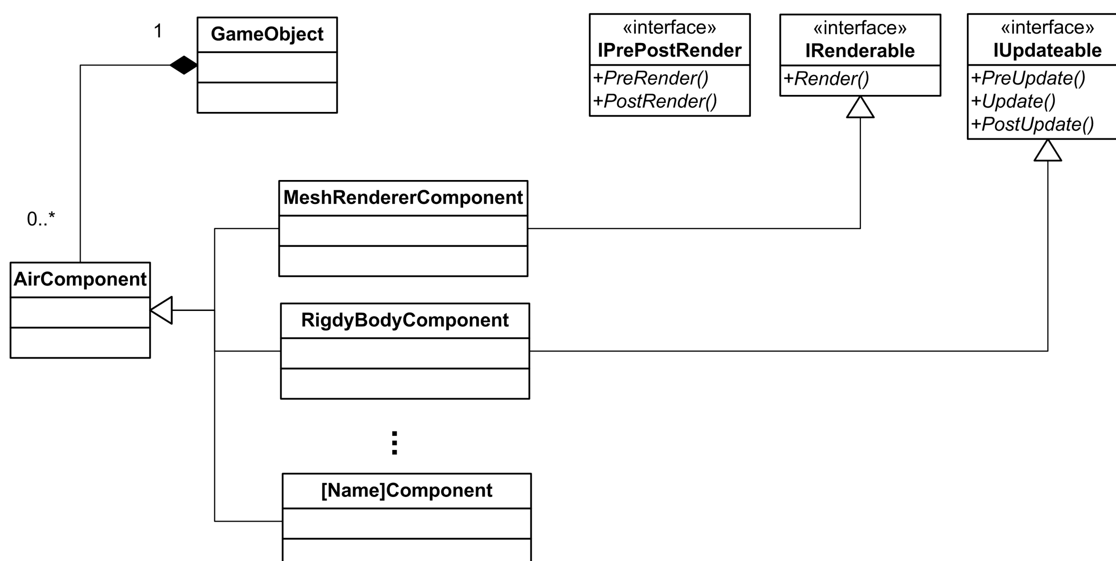

Figura 5: Interfaces

Cada componente debe registrarse a sí mismo para poder recibir los eventos. Esto es parte de la propiedad de autosuficiencia de los componentes. Más adelante se explicará el proceso de registración.

2.2. Reflexión

Cada componente para poder ser autosuficiente, al momento de compilarlo no debe asumir nada sobre el entorno sobre el que se va a ejecutar. Es por ello cada componente debe disponer de un mecanismo para poder explorar los otros componentes pertenecientes al mismo GameObject. Esta exploración para que sea general debe realizarse por tipos y no por nombres o identificadores. Es decir un componente debe poder determinar la presencia de otros componentes de determinado tipo. Al hacer esta busqueda por tipos permite que los componentes puedan ser compilados por separado, y que agregar nuevos componentes no signifique tener que cambiar identificadores en otras partes del motor.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

La reflexión es una característica presente en algunos de los lenguajes modernos. La misma se define como la capacidad por el cual un programa de computación puede observar (realizar introspección de tipos) y modificar su propia estructura de alto nivel en tiempo de ejecución. Todos los lenguajes admitidos por .NET poseen esta característica. Usando reflexión el GameObject puede realizar búsquedas por tipos, cumpliendo el requerimiento expresado en el párrafo anterior.

Otra de las ventajas de usar reflexión es la posibilidad de hacer enlazado tardío (*late binding*). Esto permite agregar nuevos componentes durante la ejecución del simulador sin necesidad de detenerlo ni volver a compilar nada.

Gracias a estas características ya no es necesario utilizar lenguajes externos de scripting ya que cada componente puede ser examinado y modificado en tiempo de ejecución y además se puede compilar código nuevo y enlazarlo de manera transparente.

Por otro lado también presenta la posibilidad de implementar generadores automáticos de interfaces. Esto permite que no haya necesidad de estar modificando las interfaces cada vez que se crea un componente nuevo, ya que la misma se actualizará automáticamente.

2.3. Estructura global

Air está compuesto por una clase principal *AirGame* que se encarga de inicializar el sistema de componentes y maneja el lazo principal del simulador. Además es la responsable de mantener y administrar la colección de GameObjects. AirGame también se comporta como un contenedor de componentes para aquellos casos donde el componente presente una mayor jerarquía y deba ser accedido por cualquier otro componente.

AirGame es la responsable también de administrar los grupos de componentes registrados y llamar los eventos correspondientes. Cada componente debe registrarse con esta clase para empezar a recibir los eventos.

El diagrama de clases se puede observar en la Fig. 6.

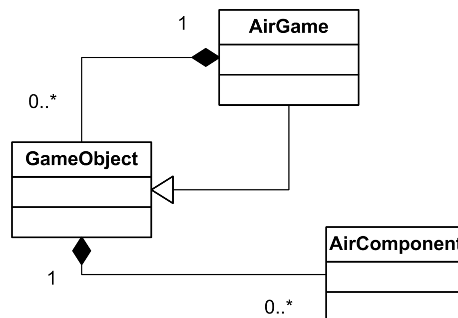

Figura 6: Relación entre AirGame y GameObject

2.3.1. Singletons

Para la creación de los objetos principales del simulador se utiliza un patrón creacional conocido como *singleton*. El mismo restringe la existencia a una sola instancia del tipo en cuestión. Además permite accederlo desde cualquier punto de la aplicación sin necesidad de pasar punteros como argumentos. Por ejemplo el objeto AirGame funciona como singleton. Para conseguir dicho comportamiento el mismo se declara como:

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
public class AirGame : GameObject
   {
       public static AirGame Instance
       {
           get
           {
               return instance;
           }
       }
       private static readonly AirGame instance = new AirGame();
       private AirGame(){ }
    }
```

instance es la única instancia que existirá a lo largo de la vida del programa de la clase AirGame. Dicha instancia se crea de manera estática al lanzar el programa y es de sólo lectura. Además, se provee la propiedad Instance que al ser estática permite ser leída desde cualquier ensamblado enlazado contra el proyecto. También debe declararse el constructor como privado para evitar múltiples instancias.

2.3.2. Contenedores de grupos

AirGame utiliza tres clases para almacenar los componentes pertenecientes a los distintos grupos: ProcessList, RenderableList y PrePostRenderList. Dichas clases también implementan el patrón singleton, simplificando la tarea de agregarle o quitarles elementos. Si se define un componente nuevo que debe actualizarse en el grupo de simulación entonces el código sería:

```
public class ComponentePrueba : AirComponent, IUpdateable
    {
        public override void Load()
        {
            base.Load();
            //se registra para el evento de update
            //usando el patron singleton
            ProcessList.Instance.RegisterUpdateable(this);
        }
        //Implementacion de la interfaz IUpdateable
        void IUpdateable.PreUpdate(){}
        void IUpdateable.Update(int dt){}
        void IUpdateable.PostUpdate(){}
   }
```

2.4. Loop principal

El loop principal de la aplicación se puede observar en la Figura Fig.7

Allí se puede observar el orden en que son llamados los grupos de contenedores. Quienes a su vez realizan las llamadas correspondientes a sus componentes registrados. A modo de ejemplo se muestra en la Fig.8 el caso para el ProcessList.

3. CASO DE APLICACIÓN

En la presente sección se propone como ejemplo la aplicación de Air a la simulación de cuerpos rígidos en tiempo real. Para ello se desarrollarán dos componentes en C++ los cuales utilizarán Physx(Nvidia, 2011) de NVidia® como librería de física.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

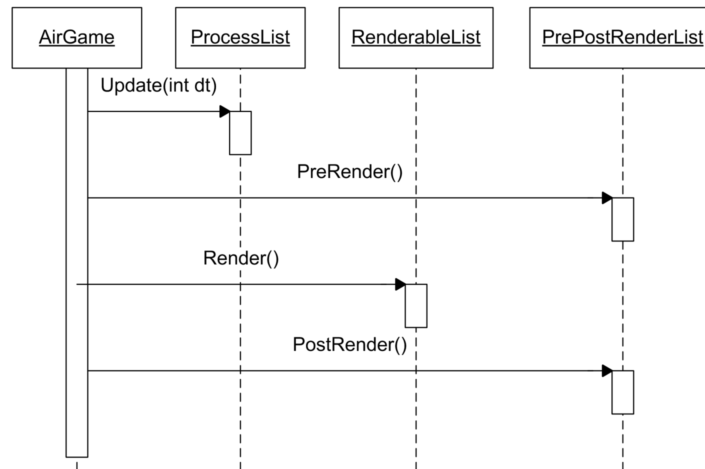

Figura 7: Diagrama de secuencia del loop principal del simulador

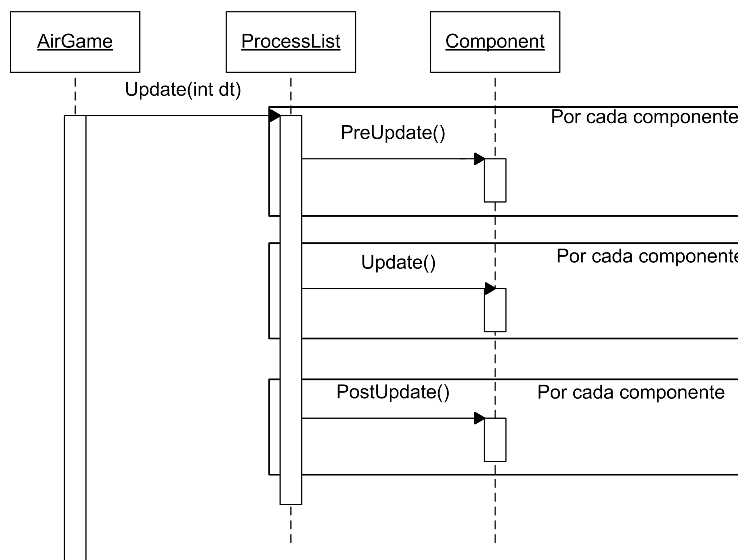

Figura 8: Diagrama de secuencia del método Update() de la clase ProcessList

3.1. El mundo

En primer lugar se desarrolló un componente que encapsula el objeto NxScene de Physx. El mismo representa el mundo simulado y es el de mayor jerarquía dentro de la simulación. Para ello se implementó el siguiente componente

```
public ref class NxWorld : public AirComponent, IUpdateable
{
public:

//Puntero para exponer el miembro nativo
    property NxScene * NativePointer{
        NxScene * get()
        {
            return _scene;
        }
    }
    NxWorld(){
        //Creacion de los objetos nativos
```

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
            //de Physx
            NxPhysicsSDKDesc desc;
            NxSDKCreateError errorCode = NXCE_NO_ERROR;
            NxPhysicsSDK *gPhysicsSDK;
            gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, NULL , new ↩
                PhysxErrorOutput(), desc, &errorCode);

            gPhysicsSDK->setParameter(NX_SKIN_WIDTH, 0.05f);

            // Create a scene
            NxSceneDesc sceneDesc;
            sceneDesc.gravity               = NxVec3(0.0f, -9.81f, 0.0f);
            _scene = gPhysicsSDK->createScene(sceneDesc);

            //Seteo material por defecto
            NxMaterial* defaultMaterial = _scene->getMaterialFromIndex(0);
            defaultMaterial->setRestitution(0.0f);
            defaultMaterial->setStaticFriction(0.5f);
            defaultMaterial->setDynamicFriction(0.5f);

        }

        virtual void Load() override{
            //Se registra el componente con la lista correspondiente
            ProcessList::Instance->RegisterUpdateable(this,false);
        }

        virtual void PreUpdate()
        {

        }

        virtual void Update(int dt)
        {

      //Llamada a los métodos nativos
             _scene->simulate((NxReal)dt);
            _scene->flushStream();
             _scene->fetchResults(NX_RIGID_BODY_FINISHED, true);

        }

        virtual void PostUpdate()
        {

        }

    private:

        NxScene* _scene;
    };
```

Este componente formará parte del Game ya que sólo debe existir uno y es de mayor jerarquía que cualquier otro de Physx.

3.2. Un cuerpo rígido

Por otro lado se desarrolló un componente que representa un cuerpo rígido dentro de la simulación. Para ello se implementó la clase NxRigidBody, de la cual sólo se muestra un par de métodos a continuación debido a su extensión. Dicha clase encapsula un objeto de tipo NxActor de Physx.

```
public ref class NxRigidBody : public AirComponent
    {
    public:
```

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
    //Cuando el componente se agrega al gameObject
    //sirve para inicializarlo
        virtual void Load() override{

            if(_static)
                InitStatic();
            else
                InitDynamic();

        }

    private:
        //si es un cuerpo rigido dinamico
        void InitDynamic(){

            //Objetos de Physx
            NxBodyDesc bodyDesc;
            NxActorDesc actorDesc;
            actorDesc.density=0.5;
            actorDesc.body          = &bodyDesc;
            actorDesc.globalPose.t  = NxVec3(_position.X, _position.Y, _position.Z);

            //Necesitamos el componetne NxWorld definido anteriormente. Por lo que
            //usando reflexion se recupera.

            NxWorld ^world= ParentGameObject->Game->Components->FindComponent<NxWorld ^>();
            //se crea el actor
            _actor=NULL;
            _actor = world->NativePointer->createActor(actorDesc);



        }

        //Si es un cuerpo estatico
        void InitStatic(){

          //Objetos de Physx
            NxActorDesc actorDesc;
            actorDesc.globalPose.t = NxVec3(_position.X, _position.Y, _position.Z);

            //Se busca el componente necesario
            NxWorld ^world= ParentGameObject->Game->Components->FindComponent<NxWorld ^>();

            //Se registra el actor
            _actor=NULL;
            _actor = world->NativePointer->createActor(actorDesc);


        }

        //Objetos nativos de Physx
        NxActor* _actor;
        NxScene* _world;
        bool     _static;
    };
```

3.3. Simulación

Usando las dos clases definidas se transcribe a continuación un breve y sencillo programa ilustrando su uso. En primer lugar se creó una nueva clase TestGame que hereda de Game y en su método de inicialización se le agregan tres componentes:

```
public class TestGame : Game
   {
```

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
        //Metodo que se llama para inicializar los
        //componentes
        public override void OnGameInit()
        {
            base.OnGameInit();

            //Se crea GameObject vacio
            GameObject gobj = new GameObject();

            //Se inicializa el componente global
            NxWorld worldComponent = new NxWorld();

            //Se agrega a la lista de componentes del Game
            Game.Instance.AddComponent(worldComponent);

            //Se crea un cuerpo rigido cubico
            NxRigidBody rb = new NxRigidBody();

            //Se configuran las figuras de colision
            NxBoxCollider box = new NxBoxCollider();

            //Seteo de posiciones y dimensiones
            AVector3 v = new AVector3();
            v.X = 20;
            v.Y = 20;
            v.Z = 20;
            AQuaternion q = new AQuaternion();
            q.X = 1.0f; q.Y = 1.0f; q.Z = 1.0f; q.W = 1.0f;

            rb.GlobalOrientationQuat = q;
            box.Dimensions = v;
            rb.CreateCollider(box);
            v.X = 20;
            v.Y = 100;
            v.Z = 20;
            rb.Position = v;
            //Se agrega el componente de RigidBody al
            //GameObject creado anteriormente
            gobj.AddComponent(rb);

            //Se crea el piso
            rb = new NxRigidBody();
            box = new NxBoxCollider();
            v.X = 300;
            v.Y = 10;
            v.Z = 100;
            //Es estatico el piso
            rb.Static = true;
            box.Dimensions = v;
            //se crea la figura de colision
            rb.CreateCollider(box);
            //Se agrega el componente al GameObject creado
            gobj.AddComponent(rb);


            //Se crea una esfera
            rb = new NxRigidBody();
            NxSphereCollider sph = new NxSphereCollider();
            v.X = 40;
            v.Y = 400;
            v.Z = 10;
            rb.Position = v;
            sph.Radius = 10.0f;
            rb.CreateCollider(sph);
            //Se agrega al mismo GameObject
            gobj.AddComponent(rb);
        }
    }
```

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

En este ejemplo se crean 3 cuerpos rígidos y se agregan al mismo GameObject. Se crean dos cajas, de las cuales una funciona como el piso y es estática. También se crea una esfera que caerá por encima de la caja y debe deslizarse hasta salir del plano del piso.

Luego, en el punto de entrada de la aplicación:

```
class Program
  {
      static void Main(string[] args)
      {
          TestGame.Instance.Loop();
      }
  }
```

Utilizando el depurador remoto de Nvidia® se puede observar la secuencia mostrada en la Fig. 9. Allí se puede muestra como la caja se posa sobre el plano del piso y luego la esfera cae sobre la misma, rebota y rueda hasta salirse del plano del piso.

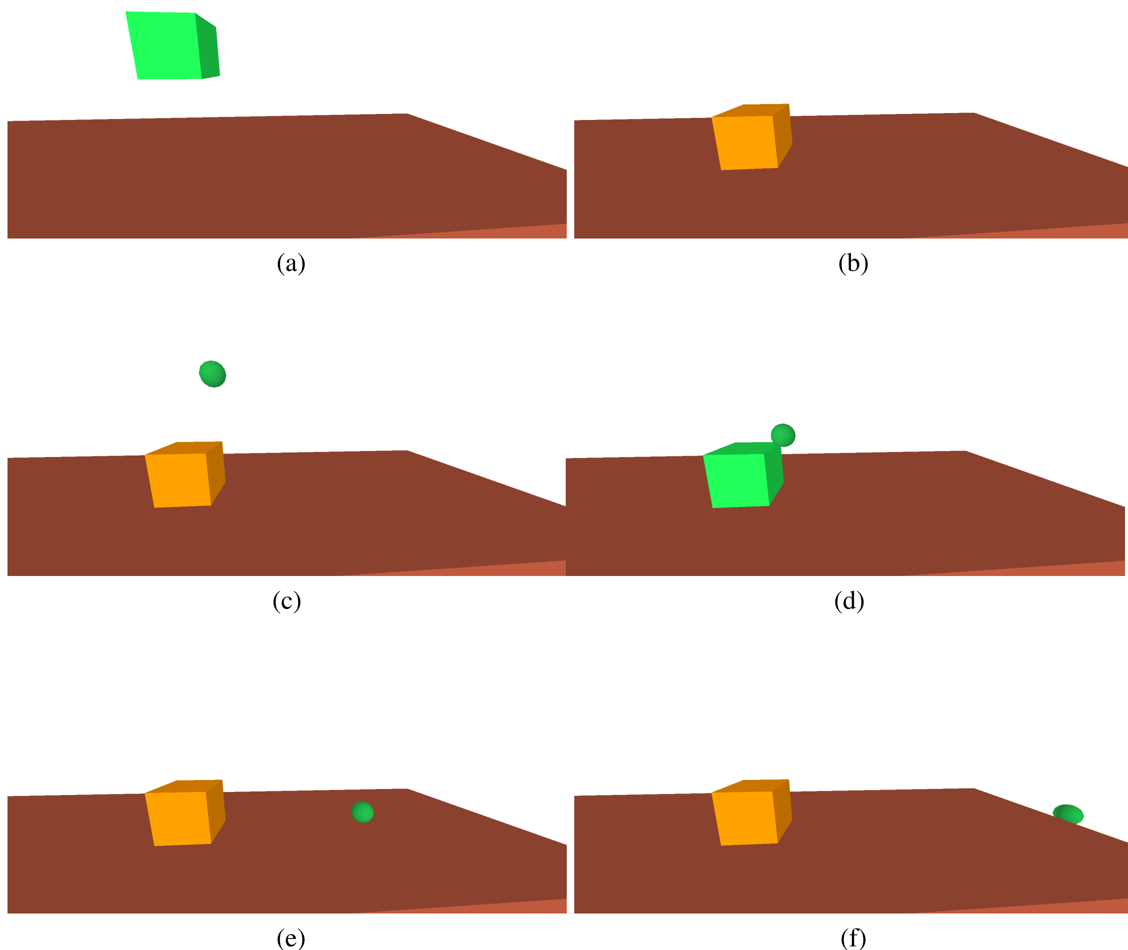

Figura 9: Secuencia ejemplo

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

4. CONCLUSIONES

En el presente trabajo se presentó un nuevo diseño orientado a componentes basado en reflexión para simuladores de propósito general. El mismo es suficientemente flexible y extensible para ser utilizado en todo tipo de proyectos. Se introdujo el diseño utilizado para la implementación del mismo y se detallaron las ventajas que posee sobre el enfoque tradicional de orientación a objetos. Entre las características más sobresalientes del mismo se debe resaltar la posibilidad de modificarlo enteramente en tiempo de ejecución utilizando reflexión. Esto incluye quitar y agregar componentes, modificar sus valores, etc. sin necesidad de implementar ninguna capa de scripting adicional. También permite el trabajo concurrente en diferentes lenguajes de manera transparente. Esto permite, por ejemplo, reutilizar código ya desarrollado en C++, como se mostró en el ejemplo dado de Physx.

Actualmente *Air* está siendo utilizado para desarrollar la nueva versión del simulador de vuelo Excalibur (Limache A.C. , Rojas Fredini P.S. y Murillo M.H., 2010) mostrando resultados prometedores.

REFERENCIAS

Deltaknowledge. Chrono::Engine. http://www.chronoengine.info/chronoengine/index.html, 2011.

Epic Games Inc. Unreal Development Kit. http://www.udk.com/, 2011.

Garage Games. Torque. http://www.garagegames.com/, 2011.

García Bauza C. , Boroni G. , Vénere M. , Clausse A. Realtime Interactive Animations of Liquid Surfaces with Lattice-Boltzmann Engines. *Aust. J. Basic & appl. Sci.*, páginas 3730–3740, 2010.

García Bauza C., Lotito P. , Parente L. , Vénere M. Incorporación de comportamiento físico en motores gráficos. In *Mecánica Computacional*, volumen XXVII, páginas 1247–1258. 2008.

Lazo M. , García Bauza C. y Clausse A. Animación de tornados en tiempo real mediante motores físicos. In García Bauza C., Lotito P., Parente L., Vénere M., editor, *Mecánica Computacional*, volumen XXVIII, páginas 1247–1258. 2009.

Limache A.C. , Rojas Fredini P.S. y Murillo M.H. Diseño de un moderno simulador de vuelo en tiempo real. In AMCA, editor, *MECOM*. 2010.

Microsoft. .Net Framework. http://www.microsoft.com/net/, 2011.

Nvidia. Physx. http://www.nvidia.com/object/physx_new.html, 2011.

Nystrom R. Component Programming Pattern, 2010.

Rojas Fredini P.S. , Limache A.C. Simulación de Fluidos en Tiempo Real Usando SPH. In Dvorkin E., Goldschmit M., Storti M., editor, *Mecánica Computacional*, volumen XXIX. 2010.

Tasora A. , Anitescu M. . A fast ncp solver for large rigid-body problems with contacts, friction and joints. *Multibody Dynamics*, 12:252, 2009.

The C++ Standards Committee. C++ standard. http://www.open-std.org/jtc1/sc22/wg21/, 2011.

Unity Technologies. Unity 3. http://unity3d.com/, 2011.

Copyright © 2011 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar