UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

EXPERIMENTANDO CON VIDEOJUEGOS EN XNA:
MANUAL DE LABORATORIO

REPORTE DE ACTIVIDAD DOCENTE

QUE PARA OBTENER EL TÍTULO DE:

MATEMÁTICO

P R E S E N T A :

APOLO OSORNIO OSORNIO

Directora de Tesis: Ana Luisa Solís González-Cosío

2007

FACULTAD DE CIENCIAS

Experimentando con Videojuegos en XNA: Manual de Laboratorio

Hoja de Datos del Jurado

1. Datos del alumno. Autor. Apellido Paterno: Apellido Materno: Nombre: Teléfono: Universidad: Facultad o escuela: Carrera: No. de cuenta:	1. Datos del alumno. Osornio Osornio Apolo 53360400 Universidad Nacional Autónoma de México Facultad de Ciencias Matemáticas 09232620-4
2. Datos del tutor. Apellido Paterno: Apellido Materno: Nombre:	2. Datos del tutor. Solís González-Cosío María Concepción Ana Luisa
3. Datos del sinodal 1. Apellido Paterno: Apellido Materno: Nombre:	3. Datos del sinodal 1. Solís González-Cosío María Concepción Ana Luisa
4. Datos del sinodal 2. Apellido Paterno: Apellido Materno: Nombre:	4. Datos del sinodal 2. López Mendoza Salvador
5. Datos del sinodal 3. Apellido Paterno: Apellido Materno: Nombre:	5. Datos del sinodal 3. Galicia Haro Sofía Natalia
6. Datos del sinodal 4. Apellido Paterno: Apellido Materno: Nombre:	6. Datos del sinodal 4. Oktaba Hanna
7. Datos del sinodal 5. Apellido Paterno: Apellido Materno: Nombre:	7. Datos del sinodal 5. López Gaona Amparo
8. Datos del trabajo escrito. Título: Subtítulo: No. de páginas: Año:	9. Datos del trabajo escrito. Experimentando con XNA: Manual de Laboratorio 126 p. 2007

Agradecimientos

A mi familia, por su gran cariño: a mis padres Alfredo y Amalia, a mis hermanos Tona y Edith, a mi cuñada Aida, a mi sobrina Eyra, que vino a dar alegría a nuestras vidas, a cambiarlas... A ellos, que siempre me han apoyado y han creido en mis proyectos por inviables que estos parezcan; es imposible agradecer las vivencias y los valores que me dieron.

Al pequeño Chosmky, por su cariño incondicional.

A Fabiana, por su gran amor, por su paciencia, por sus palabras de aliento, por llegar a mi vida...

A Ana Luisa, por dirigir mi tesis, por su gran amistad, por el apoyo que me brindó más de lo necesario para el desarrollo de este trabajo. ¡¡Gracias, Ana!!

A todos mis amigos, a los integrantes del cubículo (Susana, Tania, Reinel); a Lourdes Guerrero; a Zinnia, Niko, May, Turena, Edmundo, Lucía Mendoza; a Jorge y Lucía Doval; a mi gran amigo David Guerrero; a Carlos Serrato; y a algunos que se encuentran muy lejos: Mauricio, Chava, Ricardo, Maricarmen, Eugenia De Combi, Juan Manuel... A todos mis amigos, aunque no estén mencionados aquí, ¡gracias!

A mi tío Armando y a su familia, por su cercanía y apoyo a lo largo de mi vida.

A Ana Irene Ramírez, por mostrarme la gran persona que es, su amistad, sus consejos... Y gracias también a su grupo de trabajo: Juan Pablo, Santiago y Felipe.

Al M. en C. Agustín Ontiveros, por la orientación en este proceso.

A mis sinodales – Mat. Ana Luisa Solís, Mat. Salvador López, Dra. Sofía Galicia, Dra. Hanna Oktaba, Dra. Amparo López -, por su disposición, tiempo y orientación.

A mi Universidad, por darme una visión diferente del mundo - una visión humanista-, por darme el espacio y la oportunidad para mi crecimiento.

A mi México, el lugar al que siempre puedo regresar, del que puedo despegar, el que me da las fuerzas para luchar, para buscar motivación de cambio, al que puedo mejorar, por ser simplemente mi hogar y haberme dado todo.

Tabla de Contenido

INTRODUCCION

Debido a la necesidad de formar personas que se puedan integrar de forma idónea a la industria de videojuegos, buscamos la manera de integrar un programa para su desarrollo en la Facultad de Ciencias, en colaboración con Microsoft, cuyo resultado fue la creación de prácticas de laboratorio para la enseñanza de videojuegos.

Estas prácticas fueron elaboradas sobre la investigación, experimentación e interacción que se obtuvo con los estudiantes en el Seminario de Videojuegos impartido en el semestre 2007-II en esta Facultad.

La plataforma que seleccionamos para el desarrollo de videojuegos para PC y Consolas XBOX 360 es XNA Game Studio Express, la cual fue liberada en noviembre del 2006 y, en conjunto con el lenguaje de programación C#, sirve para desarrollar videojuegos de manera más fácil y eficiente, acercando esta actividad a un amplio abanico de personas.

El desarrollo de un videojuego requiere de una formación sólida en las áreas de programación, matemáticas, inteligencia artificial, física, cómputo gráfico, diseño, entre otras , y puede resultar un proceso complicado, así que es necesaria la elaboración de material de ayuda para el estudiante.

La elección de los temas tratados en este material va orientada por la necesidad de aclarar los temas básicos en la elaboración de un videojuego. La secuencia con la que son presentados obedece al grado de dificultad que contienen los temas, y éstos se vuelven más complejos progresivamente, a medida que el lector avanza.

En la práctica 1, hablamos de cómo organizar las ideas para crear un juego, tomando en cuenta sus obstáculos, niveles de dificultad y objetivos; también de la importancia del diseño para el éxito del proyecto. En la 2, hacemos una breve introducción al XNA, desde la instalación de un laboratorio a la configuración del entorno y la creación de un primer proyecto. En la práctica 3, presentamos los modeladores más populares para la elaboración de objetos y escenarios 3D. En las prácticas 4 y 5, incorporamos, respectivamente, los dispositivos de mando y de audio necesarios para lograr la interactividad con el usuario. En la práctica 6, tratamos sobre la creación y manejo de cámaras, las cuales utilizamos para ofrecerle al usuario la facilidad de personalizar o

manipular la visión de juego. En la 7, usamos la herramienta Skybox como recurso para optimizar el entorno y delimitar el espacio de juego. En la práctica 8, implementamos herramientas de simulación y leyes físicas, que le permiten al usuario una mejor experiencia. En la 9, presentamos una guía de cómo crear una interfaz, la cual se encarga de darnos la información y la interacción sobre el juego. Aunque éste no es un libro de modelado de personajes, en la práctica 10 hacemos un pequeño tutorial del tema. Finalmente, en la práctica 11, abordamos el tema de huesos en XNA, con el que daremos vida al personaje.

Cada práctica en este material contiene una introducción al tema a manera de tutorial y su finalidad es servir de guía al alumno para poder aprender los elementos fundamentales en el desarrollo de un juego, desde su nivel más básico hasta el funcionamiento del entorno de manera gradual.

Como objetivo final, pretendemos que el alumno tenga los elementos suficientes para aplicar las herramientas antes vistas en la elaboración de un videojuego, lo que será el proyecto final.

NOTAS AL LECTOR:

- Para algunas prácticas es necesario el uso de archivos que podrás encontrar en un CD anexo.

- Los videojuegos que sean desarrollados sobre el entorno XNA en la computadora pueden ser exportados a la consola Xbox 360. Para lograrlo, incluimos en el último capítulo los pasos necesarios.

- La plataforma XNA es gratuita (ver práctica 1), mientras que para el uso del modelador se debe adquirir una licencia, que dependerá de la elección del usuario o grupo, según convenga.

Práctica

1

NOMBRE DEL ALUMNO:	
INSTRUCTOR:	
FECHA:	
FASE:	
VIDEOJUEGOS CON XNA	

Diseño de Videojuegos

Objetivos:

- Diseño de un juego
- Objetivos, análisis y concepción de un juego
- Simulación y retroalimentación
- Presentación de un proyecto

Introducción

El trabajo del diseñador consiste en crear objetivos, reglas y procesos, es el encargado de planear todo lo necesario para crear una buena experiencia de juego al usuario.
Las habilidades que debe tener el diseñador son sobre todo una gran capacidad y pasión por comunicarse, debe mostrar pasión por los retos y el trabajo en equipo, ser altamente creativo, tener una gran necesidad de experimentar y sobre todo ser poseedor de una gran paciencia.

El proceso de diseñar un videojuego puede parecer trivial, y es por eso que a veces subestimamos esta fase. Sin embargo es quizás el proceso más importante, ya que la concepción del juego requiere de un trabajo de planeación muy fuerte y existe la necesidad de formalizar lo que hemos sólo pensado. Así que antes de encender una computadora tenemos que agarrar lápiz y papel y comenzar a experimentar, a probar todas las formas posibles, establecer los retos, definir los costos, conocer los géneros, su estructura y su desarrollo, hacer las pruebas necesarias al funcionamiento del juego y sobre todo comunicarnos con el usuario.

Por supuesto para conocer toda la teoría sobre diseño de juegos no sería posible resumirlo en unas cuantas líneas, es por eso que antes de empezar con esta serie de prácticas te recomendamos como primera tarea la lectura de *Game Design Workshop,* de los autores Fullerton, Swain y Hoffman, donde podrás encontrar talleres complementarios a los que hay aquí.

Material

- Lápiz y papel
- Monedas, fichas, botones etc

Desarrollo

Laboratorio #1
• Diseña un juego que pueda ser utilizado en clase con objetos de uso diario (reglas, monedas, etc) • Establece las reglas y juega con los demás en clase.

Ejercicio

- Elige un juego que no sea de computadora y juégalo con un grupo de amigos.
- Analiza el juego utilizando los tópicos discutidos en clase. Escribe una reseña sobre el juego.

Laboratorio #2 Objetivos

- Dentro de un grupo pequeño, diseña un juego con al menos 3 objetivos diferentes. Puede ser un objetivo para los primeros niveles, uno para los niveles intermedios y uno más para los finales. Pon por escrito las reglas y prueba el juego con los demás participantes.

Tarea

- Comienza a trabajar en un proyecto final. Busca un tema apropiado para el juego y escribe una breve descripción sobre la idea.

Laboratorio #3 Toma de Decisiones

- Divídanse en pequeños grupos y diseñen un juego que requiera que los jugadores tomen una decisión entre dos diferentes opciones. La elección puede llevar al jugador hacia la victoria. Pongan por escrito las reglas y prueben el juego con los demás participantes.

Tarea

- Usando las técnicas discutidas en clase escribe un bosquejo de las reglas para el proyecto final.

Laboratorio #4 Competencia

- En pequeños grupos, diseña un juego donde los participantes trabajen en equipo para lograr objetivos.
- Con las mismas reglas, modifica el juego de tal manera que compitan unos contra otros.

Laboratorio #5 Manejo de Recursos

- Diseña un juego con al menos 3 recursos. Cada recurso debe contener una única manera de ganar el juego. Estos recursos pueden ser combinados para probar nuevas posibilidades ganadoras.

Tarea

- Trabaja en el proyecto final y posteriormente pruébalo con tus compañeros de grupo.

Laboratorio #6 Propuestas de Juego

- Los estudiantes presentarán su proyecto final al resto de la clase.

Laboratorio #7 Prueba de Juego

- Probaremos los juegos y haremos una critica y retroalimentación al diseñador

Tarea

- Prueba tu proyecto final con tus amigos y miembros de familia. Toma notas acerca de su interés, de la dificultad y los comentarios que hagan. Usando tus notas, escribe un documento que describa los cambios que pudieras hacer a tu juego. Explica por qué.

Ejercicio: Juegos de Rol y fichas

- Haz un juego a partir de una colección de objetos (frijoles, botones etc). Elige una ficha que represente al jugador. Juega con otros compañeros de clase.

Tarea

- Toma un juego ya existente (Monopoly, Clue etc) y plantea un nuevo juego (con reglas nuevas) con las mismas piezas.

Tarea

- Escribe una segunda versión sobre las reglas del proyecto final. Incluyendo ilustraciones.

Proyecto Final
Presentación de proyectos

Ejercicio:

- Juega y critica el proyecto final. El diseñador no deberá permitir cambiar las reglas, crear disputas o cuestionamientos. En su lugar, los jugadores deberán contar con el reglamento de juego.

Tarea

- Modifica tu juego basado en la retroalimentación y juégalo con tus amigos.

Ejercicio: Simulación

- Elige tu Videojuego favorito y crea una simulación usando sólo tarjetas, fichas o dados. No tendrás gráficos, efectos de sonido y dispositivos de entrada, pero ¿qué elementos del juego sobreviven a la conversión de electrónicos a papel?

Tarea

- Escribe un pequeño documento donde diseñes un juego de computadora ficticio.

Ejercicio: Prueba de juego

- Trabaja en formalizar las reglas. Mejora la interfaz de usuario, etc.

Tarea

- Termina el proyecto final. Analiza su evolución.

Ejercicio: Variedad

- Haz un juego con algunas variantes, con eventos que ocurren en cada turno y que pueden modificar las condiciones de victoria.

Tarea

- Actualiza las reglas del proyecto final y asegúrate de que el juego esté listo para que otros lo jueguen.

Presentación Final

Ejercicio: Juego de proyectos

- A continuación pondremos a prueba los proyectos finales.

Preguntas de Control

1. Explica cómo el conflicto puede ser creado en un juego del tipo Mario Bross.

2. ¿Cuál es el objetivo en este mismo juego?

3. ¿Cómo puedes lograr que un juego sea "exitoso"?

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Fullerton T. , Swain C., Hoffman S. **Game Design Workshop**. CMP Books, 2004.

Práctica

2

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Introducción a XNA

Objetivos:

- Instalación de laboratorios para Videojuegos en XNA
- Introducción a XNA
- Visión General XNA
- Configuración del Entorno
- Creación de un proyecto

Introducción

INTRODUCCIÓN A XNA

El mundo actual de los videojuegos ha crecido de manera sorprendente, pero de igual manera sus costos, su complejidad , las exigencias del mercado, las experiencias de usuario de las nuevas generaciones con las nuevas tecnologias, la búsqueda de nuevas historias, forma y variedad. Debido a este avance se necesitan herramientas altamente competitivas en cuestión de producción, y que permitan la integración de nuevos usuarios profesionales o aficionados.

Es por esto que Microsoft ha creado una herramienta basada en las bibliotecas .NET. llamada XNA, reduciendo los costos de producción y dando la opción a todo aquel que quiera incursionar en el mundo de los videojuegos y crear sus propias aplicaciones con gran eficiencia.

XNA Framework

Seguramente ya surgieron muchas preguntas sobre el tema, pero empezaremos definiendo lo que es XNA y qué alcances tenemos con ello. Basada en las bibliotecas .NET, XNA Framework está diseñado para el desarrollo de videojuegos de manera rápida y sencilla. Originalmente dirigida a principiantes, se ha convertido en una herramienta profesional y permite al usuario crear aplicaciones 2D y 3D para Windows y Xbox360.

Otras preguntas obligadas serían ¿en qué ambiente y en qué lenguaje puedo programar?

Microsoft ha lanzado de manera gratuita un IDE (integrated development environment) para desarrollo de juegos llamado **XNA Game Studio Express.** A través de "starter kits" nos permite de manera más directa incursionar en el mundo de los juegos olvidándonos de crear ventanas, apuntadores y concentrarnos exclusivamente en la creación del código.

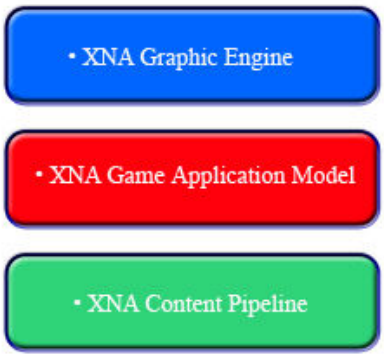
XNA fue creado para usar C#, con la idea de sustituir a MDX, y no puede ser utilizado con otros lenguajes tales como C++.

Existe una nueva versión **XNA Game Studio Profesional**, que estará disponible a mediados del 2007 y está dirigida a desarrolladores profesionales.

Modelo de Aplicación

El modelo de aplicación consiste en un grupo de controladores que simplifican de manera significativa los problemas de plataforma. Cada uno de estos contiene todos los componentes de juego que necesitamos.

Básicamente XNA Framework se puede dividir en:



El que más nos interesa es justamente el XNA Content Pipeline, que es el encargado de compilar, cargar objetos, *shaders*, sonidos, etc.

Adicionalmente, dentro del XNA Content Pipeline se permite incorporar *Componentes de juego*, que en pocas palabras consiste en "re-utilizar" módulos hechos anteriormente por otras personas.

La programación orientada a objetos define los programas en términos de *clases de objetos*, éstos colaboran entre sí para realizar tareas, permitiendo hacer en XNA los programas y módulos más fáciles de escribir, mantener y reutilizar.

Instalación de laboratorio

Requerimientos:

Los requerimientos mínimos para poder instalar el **XNA Game Studio Express** son:

- Windows XP SP2 ó Windows Vista

- Funciona desde 256 MB Ram, sin embargo se recomienda para un mejor desempeño 512 MB Ram.
- 1 GHz en CPU
- Tarjeta de video que soporte Shader Model 1.1 (Se recomiendan tarjetas de video con Shader Model 2.0, 3.0 ó bien 4.0 para mejor rendimiento, tales como GeForce, o ATI en sus series más avanzadas).

Instalación

La instalación no representa mayor problema. Para ello, sólo es necesario instalar lo siguiente:

- Microsoft Visual Studio C# Express 2005
- Descargar el XNA Game Studio Express

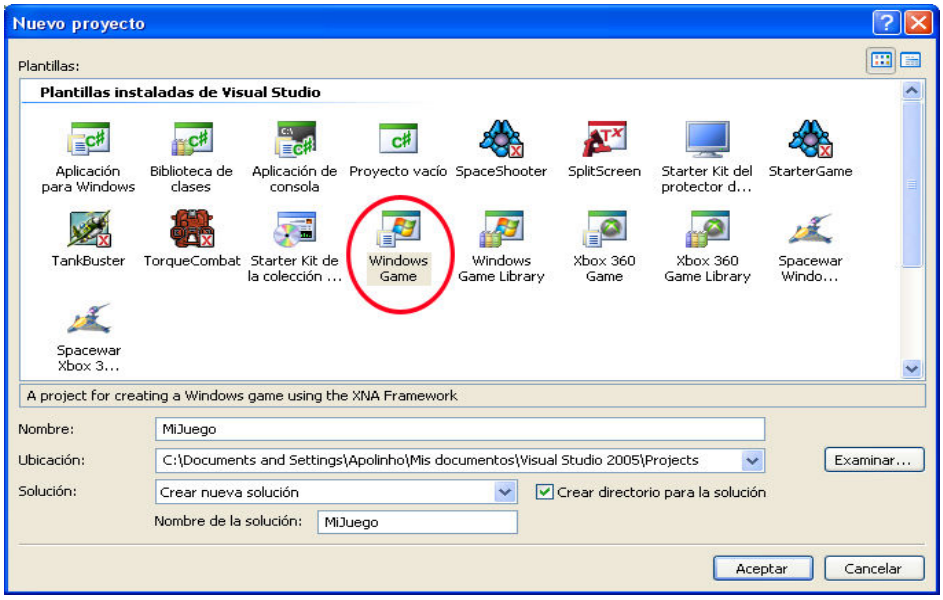
Se recomienda instalar la versión de DirectX 9.0c para el manejo de *shaders* y otras utilidades (opcional).

Una vez hecho esto, estamos listos para comenzar a programar.
En la siguiente sección únicamente crearemos un nuevo proyecto y analizaremos el código. La comprensión de éste es fundamental para entender la dinámica sobre XNA.

Primer Proyecto

Pasos:

- Abre el XNA Game Studio
- Dentro del menú principal elige **Archivo → Nuevo Proyecto**
- Aparecerá una ventana como mostramos a continuación:



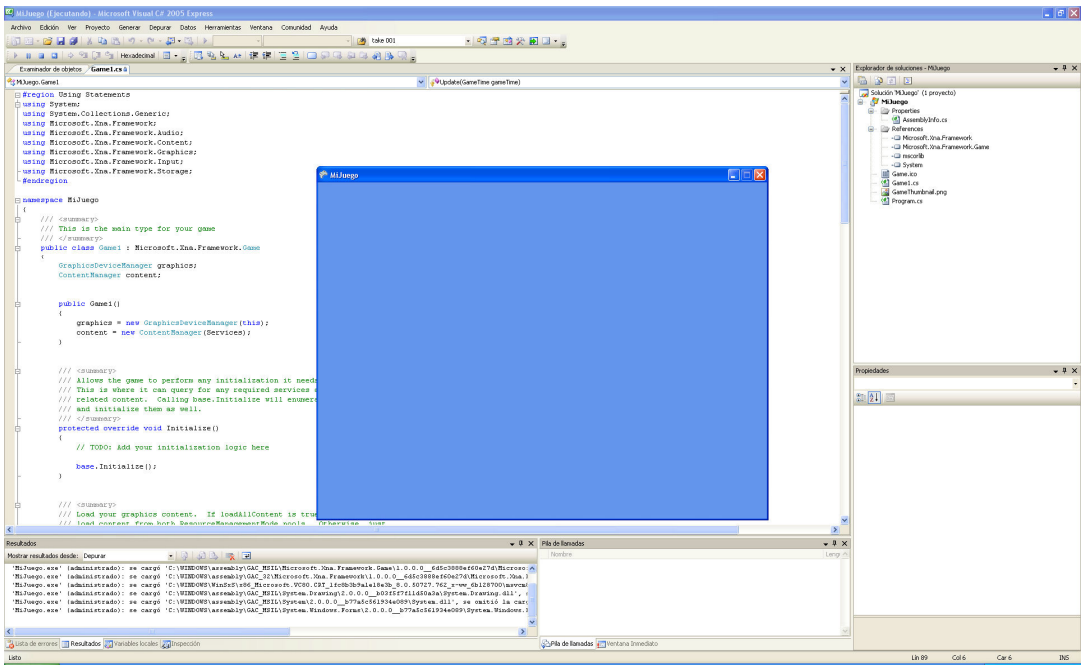
- Nombra el proyecto. Por default viene como "WindowsGame#", en este caso lo nombraremos como "Mijuego"
- Da doble clic en la opción "Windows Game"
- En este momento se ha creado nuestro proyecto con el siguiente código:

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion

namespace MiJuego
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        ContentManager content;


        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            content = new ContentManager(Services);
        }


        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to
run.
```

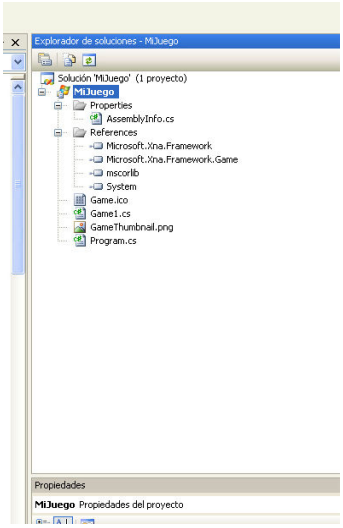
```
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any
components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }


        /// <summary>
        /// Load your graphics content.  If loadAllContent is true, you should
        /// load content from both ResourceManagementMode pools.  Otherwise, just
        /// load ResourceManagementMode.Manual content.
        /// </summary>
        /// <param name="loadAllContent">Which type of content to load.</param>
        protected override void LoadGraphicsContent(bool loadAllContent)
        {
            if (loadAllContent)
            {
                // TODO: Load any ResourceManagementMode.Automatic content
            }

            // TODO: Load any ResourceManagementMode.Manual content
        }


        /// <summary>
        /// Unload your graphics content.  If unloadAllContent is true, you should
        /// unload content from both ResourceManagementMode pools.  Otherwise, just
        /// unload ResourceManagementMode.Manual content.  Manual content will get
        /// Disposed by the GraphicsDevice during a Reset.
        /// </summary>
        /// <param name="unloadAllContent">Which type of content to unload.</param>
        protected override void UnloadGraphicsContent(bool unloadAllContent)
        {
            if (unloadAllContent)
            {
                // TODO: Unload any ResourceManagementMode.Automatic content
                content.Unload();
            }

            // TODO: Unload any ResourceManagementMode.Manual content
        }


        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }


        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
```

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

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
    }
}
```

- Si presionamos la tecla F5 obtenemos una pantalla azul como la de la imagen.



Ahora veamos nuestro entorno. Sobre la ventana superior derecha encontramos el Explorador de Soluciones, ahí tenemos todos los elementos involucrados en nuestro juego. Si observas con cuidado, existen 2 archivos.



Game1.cs y Program.cs

Uno de ellos, el *Game1.cs*, corresponde al código visto anteriormente, mientras que el *Program.cs* básicamente controla el programa.

```
using System;

namespace MiJuego
{
   static class Program
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      static void Main(string[] args)
      {
         using (Game1 game = new Game1())
         {
            game.Run();
         }
      }
   }
}
```

Como ya habíamos comentado, a diferencia de Direct X y otros entornos, XNA contiene un *starter kit* que genera automáticamente la ventana, es decir que con XNA sólo nos preocuparemos por programar nuestro juego y esto representa una gran ventaja.

Aunque hasta el momento lo único que hemos desplegado es una ventana azul, es de gran importancia comprender el código, ya que esto nos ayudará a tener una mayor idea de donde insertar los nuevos módulos

Analicemos el código *Game1*

```
namespace MiJuego
{
   /// <summary>
   /// This is the main type for your game
   /// </summary>
   public class Game1 : Microsoft.Xna.Framework.Game
   {
      GraphicsDeviceManager graphics;
      ContentManager content;
```

Aquí se encuentra el *Graphics Device Manager* y el *Content Manager*; el primero se encargará de administrar todos los dispositivos gráficos, mientras que el segundo lo hará con los contenidos. Posteriormente encontramos el *constructor* e inicializamos los *Manager*

```
public Game1()
      {
         graphics = new GraphicsDeviceManager(this);
         content = new ContentManager(Services);
      }
```

Hay 3 métodos en esta clase a los que habrá que poner especial atención***: Initialize, Update*** y ***Draw***

El método ***Initialize*** carga todo el contenido e inicializa lo necesario para nuestro juego

```
protected override void Initialize()
      {
         // TODO: Add your initialization logic here

         base.Initialize();
       }
```

El método ***Update*** es el encargado de actualizar en cada *frame* todos nuestros dispositivos, como posición de mouse, teclado, sonido, *game time,* etc. Posteriormente habrá que retomarlo cuando hablemos de interfaces.

```
base.Update(gameTime);
    }


    /// <summary>
    /// This is called when the game should draw itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
```

Finalmente el método ***Draw*** tiene la función de agregar todo lo que se desea dibujar, revisar cuadro por cuadro y actualizarlo en caso de ser necesario.

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

      // TODO: Add your drawing code here

      base.Draw(gameTime);
```

Como podrás observar en ***graphics.GraphicsDevice.Clear(Color.CornflowerBlue);*** podemos cambiar el color de la pantalla.

MATERIAL

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express

DESARROLLO

Ejercicio:
• Cambia el color de la pantalla a negro.

Preguntas de Control
1. ¿Cuáles serían las ventajas de trabajar con XNA sobre otras plataformas como DirectX, OpenGL , etc?

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. **XNA Game Programming**. Wiley Publishing, 2007.

- http://xna.animered.net/index.php?option=com_content&task=view&id=8&Itemid=14
- http://xna.animered.net/index.php?option=com_content&task=view&id=10&Itemid=14

Práctica

3

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Modeladores 3D para el Diseño de Escenas en Videojuegos

Objetivos:

- Introducción a modeladores
- Visión General sobre modelado

Introducción

En este sección daremos un breve recorrido a los modeladores ya que será importante crear los objetos 3D que añadiremos a nuestro juego.
En el mercado existe una gran cantidad de compañías que han lanzado sus productos, entre las más importantes están:

- Maya
- 3D Studio Max
- Lightwave
- Softimage
- Blender
- Rhinoceros
- Cinema 4d

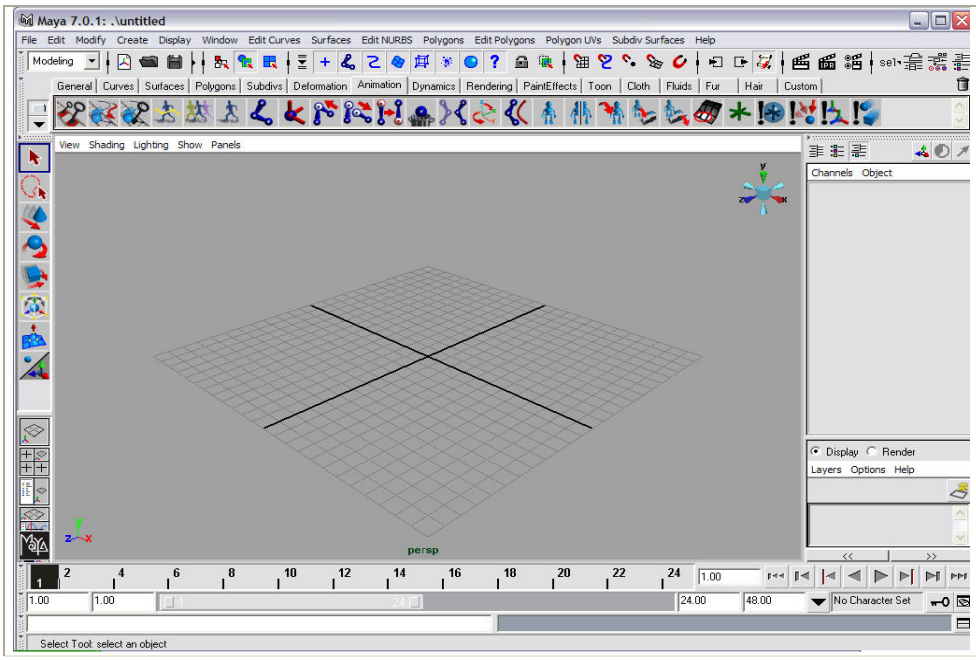
Los más populares en la industria del videojuego y de los que se puede encontrar más documentación y tutoriales son, sin duda, Maya y 3D Studio Max. Sin embargo, el usuario puede utilizar el que más se acomode a su presupuesto, gustos y necesidades, ya que existen *plugins* de conversión que pueden ser adquiridos de manera gratuita en la red, además de eso las actuales versiones manejan formatos compatibles.

Las licencias pueden variar desde las que cuestan varios miles de pesos hasta el software libre, que es el caso de Blender.

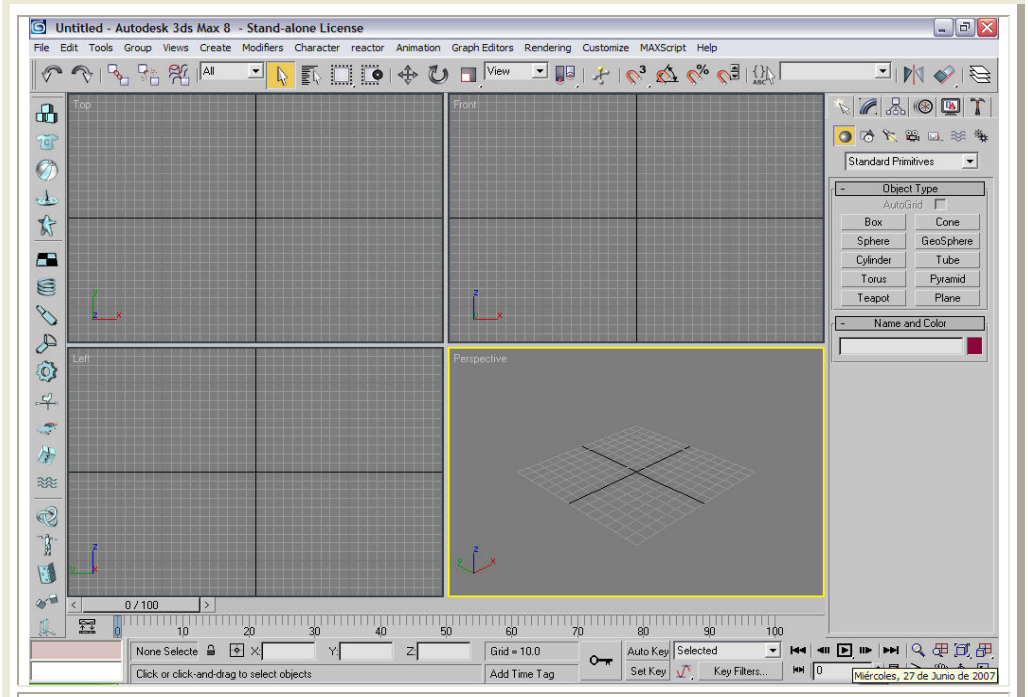
Sería inútil hacer un balance y generar un juicio de cuál es el mejor software; es importante comprender que cada uno fue creado con objetivos distintos, es decir, existen softwares especializados en modelado, como es el caso de Rhino, con el que podemos obtener más precisión. Algunos otros con técnicas que pudieran parecer más sencillas o que contengan aplicaciones o formatos más compatibles con lo que buscamos. Será tarea del alumno tomar esa decisión.

La interfaz pudiera parecer muy diferente en cada software, algunas más amigables que otras, otras más vistosas, con más botones, etc. Dar el salto entre una y otra no debe tomar mucho tiempo cuando se conocen los principios y la lógica del programa.

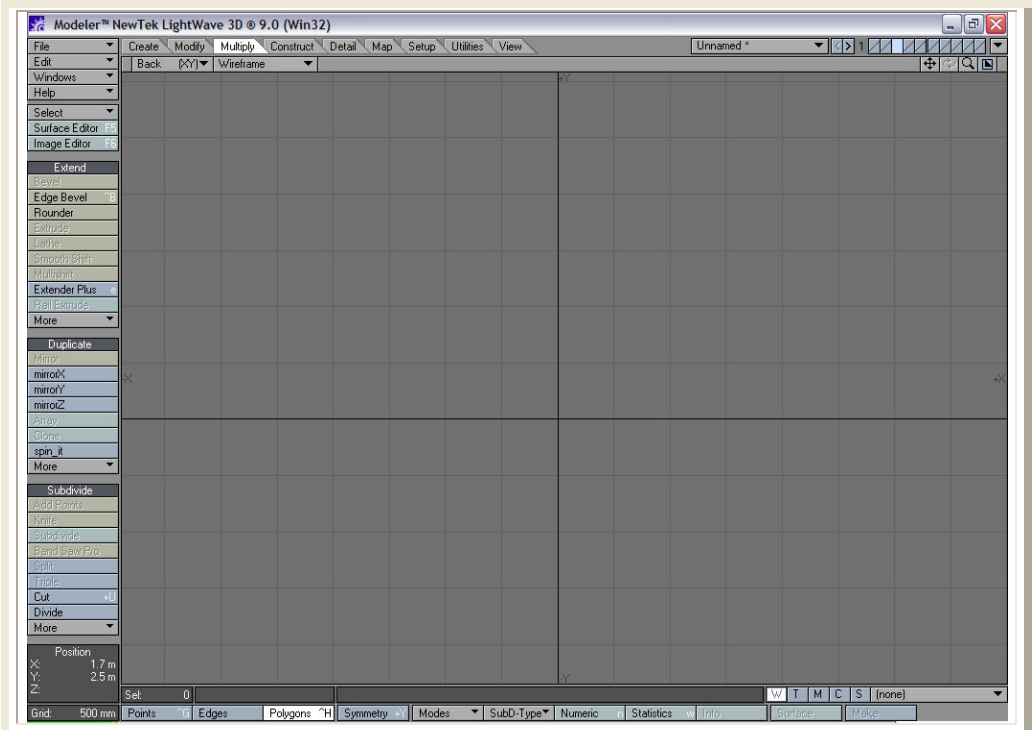
He aquí algunos ejemplos de interfaz:



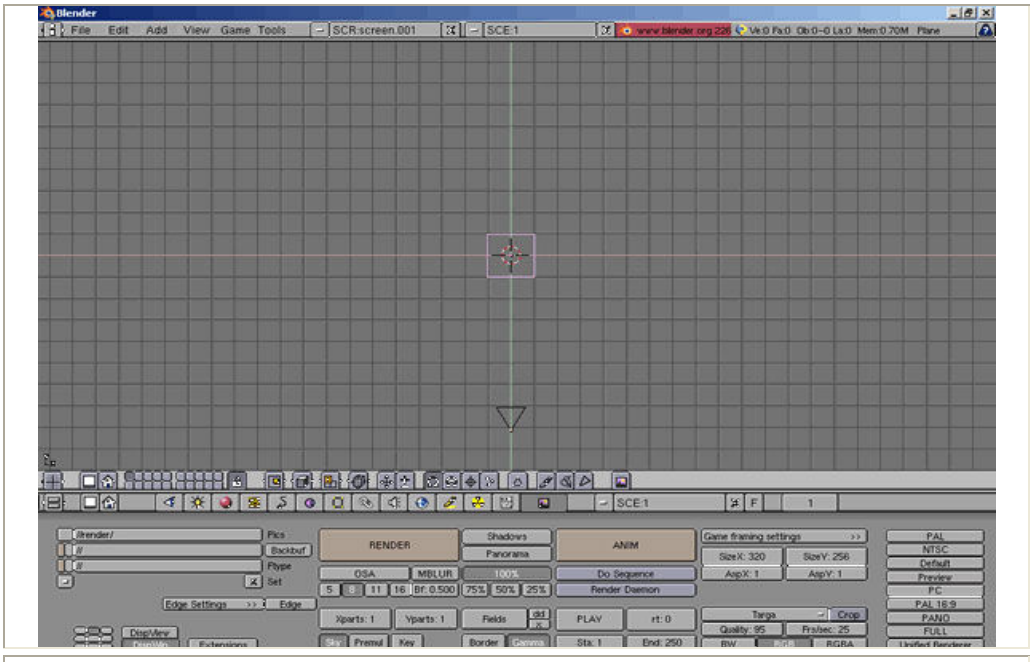
Maya



3D Studio Max



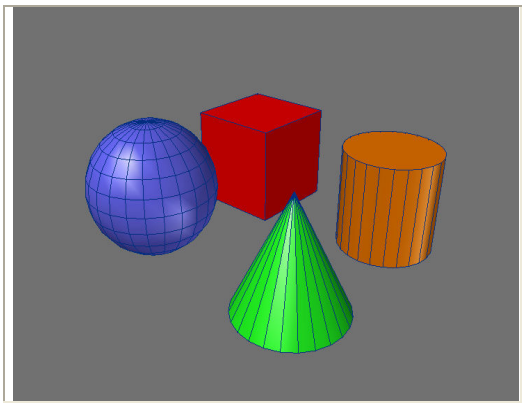
Lightwave



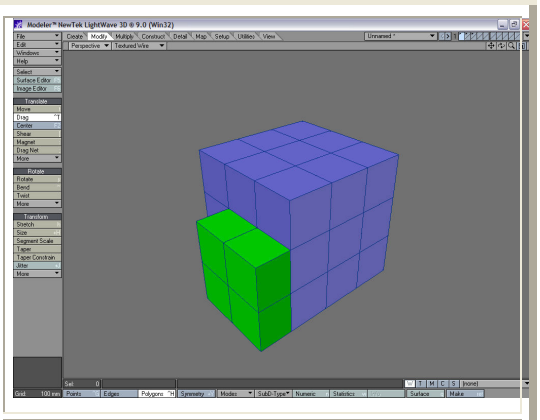
Blender

Modelado

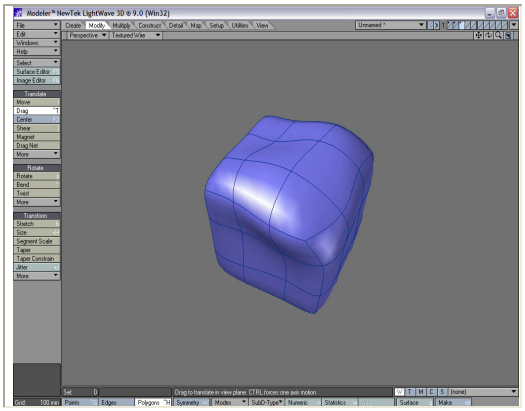
El modelado como tal consiste en darle forma al objeto. Éste puede ser creado a partir de figuras primitivas tales como un cubo, esfera, cono, cilindro etc. Con estas figuras y herramientas de nuestro modelador obtendremos geometría nueva o simplemente modificaremos la existente para dar vida a nuevos objetos.



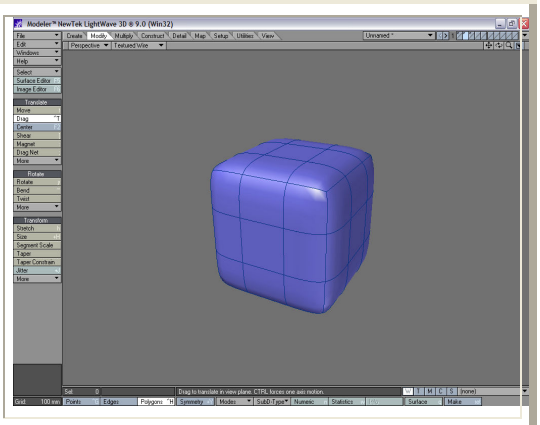
Primitivas



Obtención de geometría a partir de otra

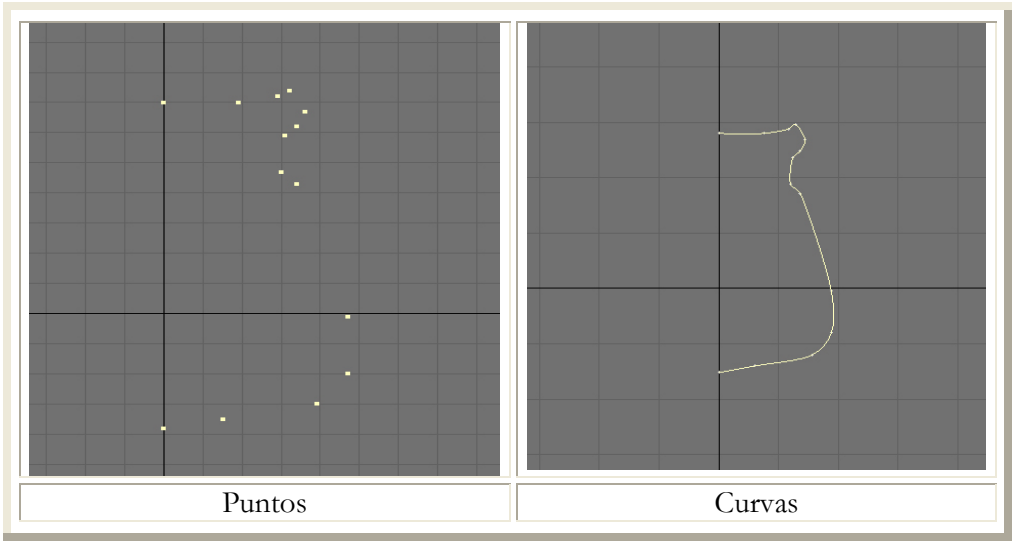


Deformaciones

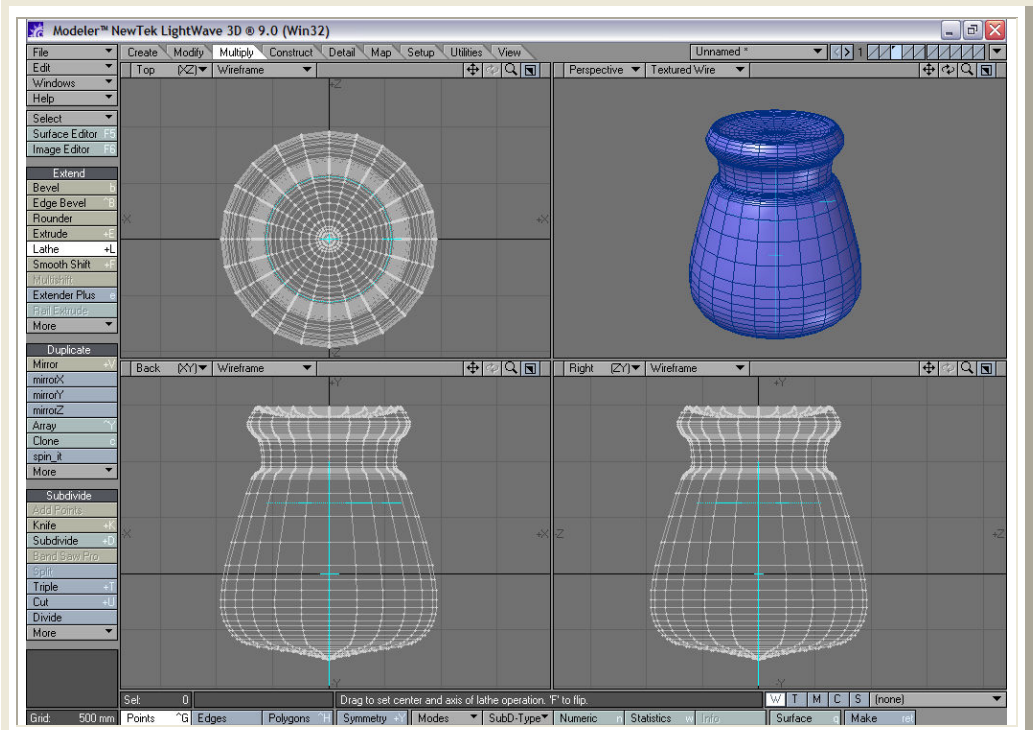


Suavizado

Podemos también crear una geometría a partir de puntos, obteniendo curvas, polígonos o sólidos de revolución.



Puntos | Curvas



Sólidos de revolución

Las posibilidades son inmensas, los límites los marcará la experiencia, la dedicación y paciencia.

Para fines de nuestro curso, es necesario que busques tutoriales que te puedan ayudar a obtener un mejor resultado.

Por otro lado, existen sitios como ***TurboSquid*** o ***3dCafe*** donde puedes adquirir los objetos ya modelados, tales como personajes, ambientes, texturas, etc. Esto puede resultar costoso y no siempre será compatible con nuestra plataforma XNA hasta después de algunas modificaciones.

Ambiente Virtual (Turbo Squid)

MATERIAL

- Modelador 3D

DESARROLLO

Ejercicio:

- Investiga sobre modeladores y explica por qué se eligió un software en particular.

Preguntas de Control

No hay preguntas de control.

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.

Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- http://www.turbosquid.com/
- http://www.3dcafe.com/

Práctica

4

NOMBRE DEL ALUMNO:	
INSTRUCTOR:	
FECHA:	
FASE:	
VIDEOJUEGOS CON XNA	

Incorporación y Manipulación de Modelos 3D utilizando un Controlador Xbox 360

Objetivos:

- En esta sección veremos cómo desplegamos un objeto 3D con texturas y cómo interactuar a través del mando de Xbox 360.

Introducción

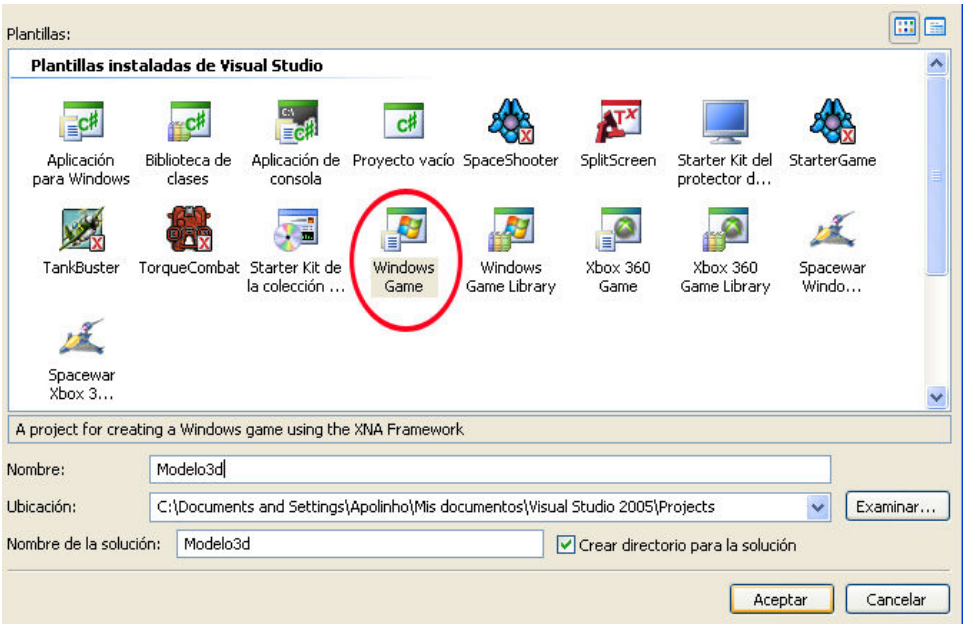
Hasta el momento hemos creado y analizado el código que nos despliega una ventana de color, pero ¿qué hay respecto a la integración de objetos en nuestro entorno y los dispositivos para interactuar?

Este tutorial intentará dar los elementos necesarios para implementar estas acciones, conoceremos acerca del *Explorador de Soluciones* y de cómo incorporar dispositivos de entrada a nuestra aplicación.

Pasos:

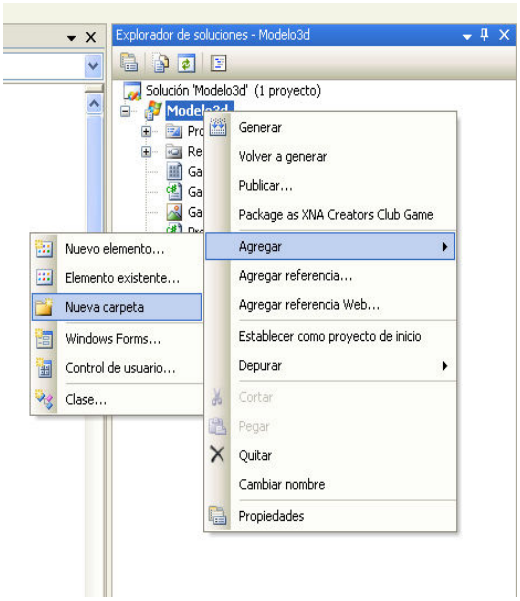
- Dentro del XNA Game Studio crea un nuevo proyecto, esto es:
  Archivo → Nuevo Proyecto

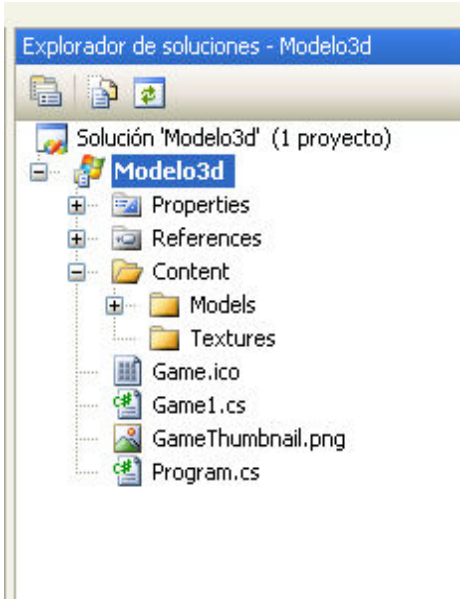
- Nombra el proyecto, lo haremos en esta ocasión como "Modelo3d"
- Da doble clic en *"Windows Game"*



Como verás se despliega el mismo código que analizamos anteriormente. El siguiente paso será incluir algunas líneas nuevas.

- Dentro del *Explorador de Soluciones* seleccionamos nuestra raíz y agregamos una nueva carpeta presionando el botón derecho del Mouse al que llamaremos "***Content***" y dentro de ésta crea 2 nuevas carpetas: "***Models***" y "***Textures***". Tendrá que quedar como en la gráfica.





Nota: Tus modelos pueden ser cargados en archivos **.fbx**, el cual es un formato de intercambio de datos 3D. En la actualidad puedes convertir cualquier tipo de formato de cualquier modelador 3D en este archivo por medio de *plugins* gratuitos que puedes obtener en la web.

Hemos facilitado modelos 3D y texturas para efectos de la práctica que están disponibles en el CD adjunto.

- Desde el CD carga el modelo ***mini.fbx*** en la carpeta *"**Models**"*.

- Posteriormente incorporaremos nuestras texturas en la carpeta del mismo nombre, disponible de igual manera en el material anexo.

Es todo lo que necesitamos en el Explorador de Soluciones, ahora vayamos al código.

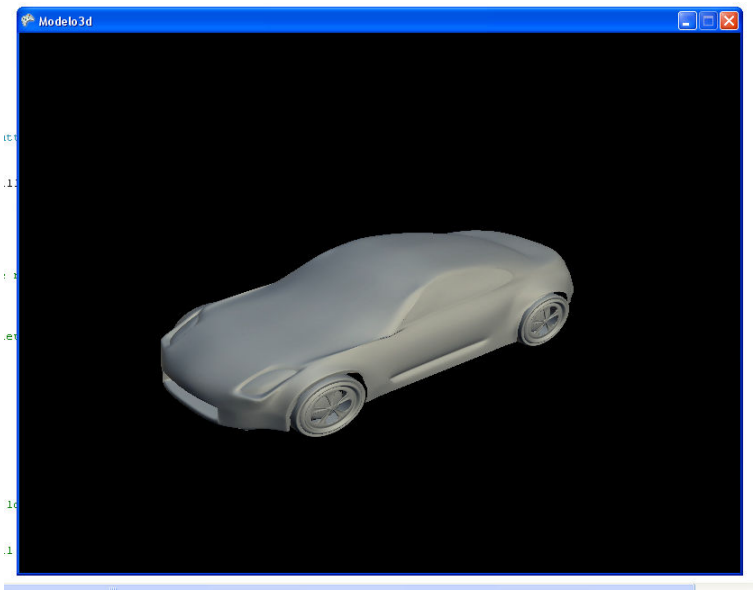
- Sustituye el código existente con lo siguiente:

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion

public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    ContentManager content;


    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        content = new ContentManager(Services);
    }

    protected override void Initialize()
    {

        base.Initialize();
    }

    // Specify the 3D model to draw.
    Model myModel;

    // The aspect ratio determines how to scale 3d to 2d projection.
    float aspectRatio;

    protected override void LoadGraphicsContent(bool loadAllContent)
    {
        if (loadAllContent)
        {
            myModel = content.Load<Model>("Content\\Models\\mini");
        }

        aspectRatio = graphics.GraphicsDevice.Viewport.Width /
            graphics.GraphicsDevice.Viewport.Height;
    }

    protected override void UnloadGraphicsContent(bool unloadAllContent)
    {
        if (unloadAllContent == true)
        {
            content.Unload();
        }
```

```
}


// Set the velocity of the model, applied each frame to the model's position.
Vector3 modelVelocity = Vector3.Zero;

protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Get some input.
    UpdateInput();

    // Add velocity to the current position.
    modelPosition += modelVelocity;

    // Bleed off velocity over time.
    modelVelocity *= 0.95f;

    base.Update(gameTime);
}

protected void UpdateInput()
{
    // Get the game pad state.
    GamePadState currentState = GamePad.GetState(PlayerIndex.One);
    if (currentState.IsConnected)
    {
        // Rotate the model using the left thumbstick, and scale it down.
        modelRotation -= currentState.ThumbSticks.Left.X * 0.10f;

        // Create some velocity if the right trigger is down.
        Vector3 modelVelocityAdd = Vector3.Zero;

        // Find out what direction we should be thrusting, using rotation.
        modelVelocityAdd.X = -(float)Math.Sin(modelRotation);
        modelVelocityAdd.Z = -(float)Math.Cos(modelRotation);

        // Now scale our direction by how hard the trigger is down.
        modelVelocityAdd *= currentState.Triggers.Right;

        // Finally, add this vector to our velocity.
        modelVelocity += modelVelocityAdd;

        GamePad.SetVibration(PlayerIndex.One, currentState.Triggers.Right,
            currentState.Triggers.Right);


        // In case you get lost, press A to warp back to the center.
        if (currentState.Buttons.A == ButtonState.Pressed)
        {
            modelPosition = Vector3.Zero;
            modelVelocity = Vector3.Zero;
            modelRotation = 0.0f;
        }
    }
}

// Set the position of the model in world space, and set the rotation.
Vector3 modelPosition = Vector3.Zero;
float modelRotation = 0.0f;

// Set the position of the Camera in world space, for our view matrix.
Vector3 cameraPosition = new Vector3(0.0f, 15.0f, 25.0f);

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

```
        // Copy any parent transforms.
        Matrix[] transforms = new Matrix[myModel.Bones.Count];
        myModel.CopyAbsoluteBoneTransformsTo(transforms);

        // Draw the model. A model can have multiple meshes, so loop.
        foreach (ModelMesh mesh in myModel.Meshes)
        {
            // This is where the mesh orientation is set, as well as our camera and
projection.
            foreach (BasicEffect effect in mesh.Effects)
            {
                effect.EnableDefaultLighting();
                effect.World = transforms[mesh.ParentBone.Index] *
Matrix.CreateRotationY(modelRotation)
                    * Matrix.CreateTranslation(modelPosition);
                effect.View = Matrix.CreateLookAt(cameraPosition, Vector3.Zero,
Vector3.Up);
                effect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
                    aspectRatio, 1.0f, 10000.0f);
            }
            // Draw the mesh, using the effects set above.
            mesh.Draw();
        }

        base.Draw(gameTime);
    }
}
```

- Presiona F5, aparecerá tu modelo tridimensional listo para ser manipulado con el control del Xbox 360.



En realidad, lo que hemos sustituido no es más que el mismo código con algunas líneas adicionales. El código sobre el modelo lo hemos puesto en verde, mientras que los registros de los dispositivos de entrada (input) están en rojo, para facilitar la búsqueda.
Analicemos con más detalle.

- En esta parte lo único que hacemos es cargar nuestro objeto 3D. Observa que añadirás este código sobre *LoadGraphicsContent,* especificando la carpeta ***Content →Models*** .

```
    // Specify the 3D model to draw.
    Model myModel;

    // The aspect ratio determines how to scale 3d to 2d projection.
    float aspectRatio;

    protected override void LoadGraphicsContent(bool loadAllContent)
    {
        if (loadAllContent)
        {
            myModel = content.Load<Model>("Content\\Models\\mini");
        }

        aspectRatio = graphics.GraphicsDevice.Viewport.Width /
            graphics.GraphicsDevice.Viewport.Height;
    }
```

- A continuación incorporamos código sobre el método ***Draw***. Aquí dibujamos nuestro modelo, indicando su posición, rotación, iluminación, etc. De igual manera, daremos información sobre la posición de nuestra cámara.

```
// Set the position of the model in world space, and set the rotation.
    Vector3 modelPosition = Vector3.Zero;
    float modelRotation = 0.0f;


// Set the position of the Camera in world space, for our view matrix.
    Vector3 cameraPosition = new Vector3(0.0f, 15.0f, 25.0f);

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

        // Copy any parent transforms.
        Matrix[] transforms = new Matrix[myModel.Bones.Count];
        myModel.CopyAbsoluteBoneTransformsTo(transforms);

        // Draw the model. A model can have multiple meshes, so loop.
        foreach (ModelMesh mesh in myModel.Meshes)
        {
            // This is where the mesh orientation is set, as well as our camera and
projection.
            foreach (BasicEffect effect in mesh.Effects)
            {
                effect.EnableDefaultLighting();
                effect.World = transforms[mesh.ParentBone.Index] *
Matrix.CreateRotationY(modelRotation)
                    * Matrix.CreateTranslation(modelPosition);
                effect.View = Matrix.CreateLookAt(cameraPosition, Vector3.Zero,
Vector3.Up);
                effect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
                    aspectRatio, 1.0f, 10000.0f);
            }
            // Draw the mesh, using the effects set above.
            mesh.Draw();
        }
```

Hasta ahora hemos integrado nuestro objeto a la escena, sin embargo falta habilitar los dispositivos para manipularlo.

Para ello, necesitaremos un controlador de Xbox 360, que nos ayudará a hacer las pruebas necesarias desde la PC, sólo hace falta conectarlo a la entrada de USB.

Como habíamos visto anteriormente, el método ***Update*** es el encargado de actualizar todas las transformaciones de nuestros objetos, asi como la posición, velocidad y toda clase de variaciones, ya sea en nuestra escena como en la aparición de algún evento.

```
// Set the velocity of the model, applied each frame to the model's position.
    Vector3 modelVelocity = Vector3.Zero;

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();

        // Get some input.
        UpdateInput();

        // Add velocity to the current position.
        modelPosition += modelVelocity;

        // Bleed off velocity over time.
        modelVelocity *= 0.95f;

        base.Update(gameTime);
    }
```

En el siguiente cuadro se observan mas cláramente las actualizaciones del usuario sobre la escena. Aquí también puedes incorporar una vibración sobre el control (*GamePad)* para simular algún efecto de realismo.

```
protected void UpdateInput()
    {
        // Get the game pad state.
        GamePadState currentState = GamePad.GetState(PlayerIndex.One);
        if (currentState.IsConnected)
        {
            // Rotate the model using the left thumbstick, and scale it down.
            modelRotation -= currentState.ThumbSticks.Left.X * 0.10f;

            // Create some velocity if the right trigger is down.
            Vector3 modelVelocityAdd = Vector3.Zero;

            // Find out what direction we should be thrusting, using rotation.
```

```
            modelVelocityAdd.X = -(float)Math.Sin(modelRotation);
            modelVelocityAdd.Z = -(float)Math.Cos(modelRotation);

            // Now scale our direction by how hard the trigger is down.
            modelVelocityAdd *= currentState.Triggers.Right;

            // Finally, add this vector to our velocity.
            modelVelocity += modelVelocityAdd;

            GamePad.SetVibration(PlayerIndex.One, currentState.Triggers.Right,
                currentState.Triggers.Right);


            // In case you get lost, press A to warp back to the center.
            if (currentState.Buttons.A == ButtonState.Pressed)
            {
                modelPosition = Vector3.Zero;
                modelVelocity = Vector3.Zero;
                modelRotation = 0.0f;
            }
        }
    }
```

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360

Desarrollo

Laboratorio

- Modela cualquier otro objeto
- Conviértelo a algún formato compatible con XNA
- Intégralo a la escena junto con sus texturas
- Programa su control
- Obtén una vibración sobre el control (*gamepad*) en un evento cualquiera, para que se active al apretar alguna tecla o botón
- Asigna un tiempo de vibración al evento

Preguntas de Control

1. ¿Qué formatos de objetos 3D son compatibles con XNA?

1. ¿Qué formatos de textura son compatibles con XNA?

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.

Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. **XNA Game Programming**. Wiley Publishing, 2007.
- ms-help://MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/GoingBeyond_ContentPipeline.htm
- ms-help://MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/GoingBeyond_Input.htm

Práctica

5

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Audio en XNA

Objetivos:

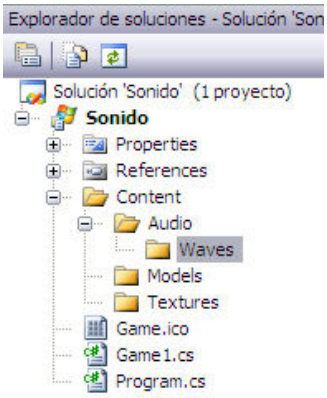
- Incorporar audio a la escena con XACT

Introducción

Audio en XNA

Una vez que hemos conseguido cargar la escena, sólo falta incluir archivos de audio, los cuales ayudarán a una mejor experiencia para el usuario. Para ello Microsoft XNA Game Studio Express contiene una plataforma que sirve como herramienta para la creación de audio llamado **XACT**.
Lo primero que necesitamos es obtener archivos de audio que queramos agregar a nuestro juego en formato *.wav*. Puedes encontrar algunos de ellos de manera gratuita en la web, pero aquí te facilitamos dos para que sigas este tutorial.
Partiremos del código anterior "práctica 4" haciendo pequeñas modificaciones.

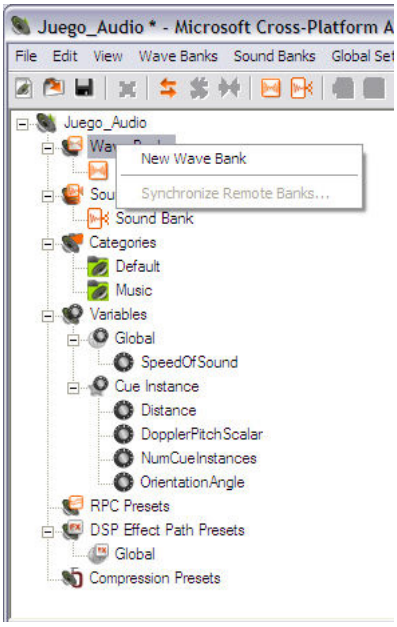
- Dentro de la carpeta *Content*, anteriormente creada, agregamos una nueva carpeta llamada *Audio* y, dentro de ella, creamos otra que se llamará *Waves*.
- Cargamos nuestros archivos .wav en la carpeta *Waves* de la misma manera que cargamos los modelos
- Está listo el Explorador de Soluciones. Ahora conozcamos la herramienta XACT.

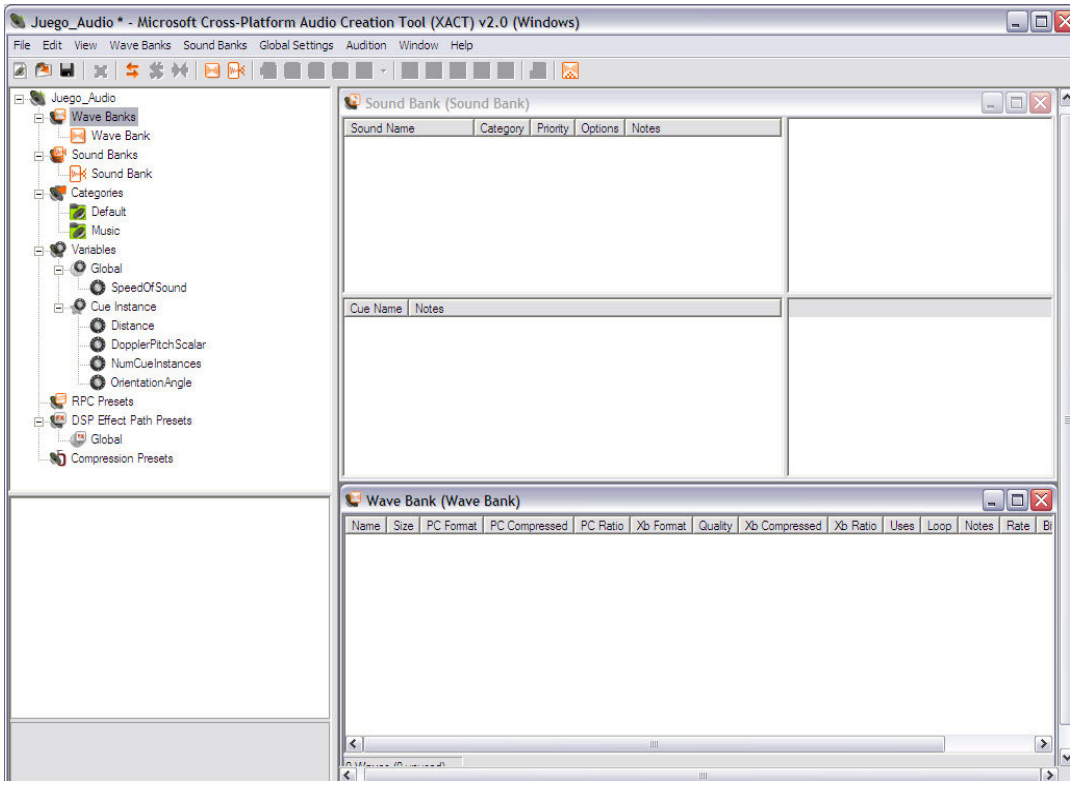


Para poder usar XACT accedemos desde el menú ***Inicio → All Programs→ Microsoft Xna Game Studio Express → Tools→ Microsoft Cross-Platform Audio Creation Tool (XACT)***.

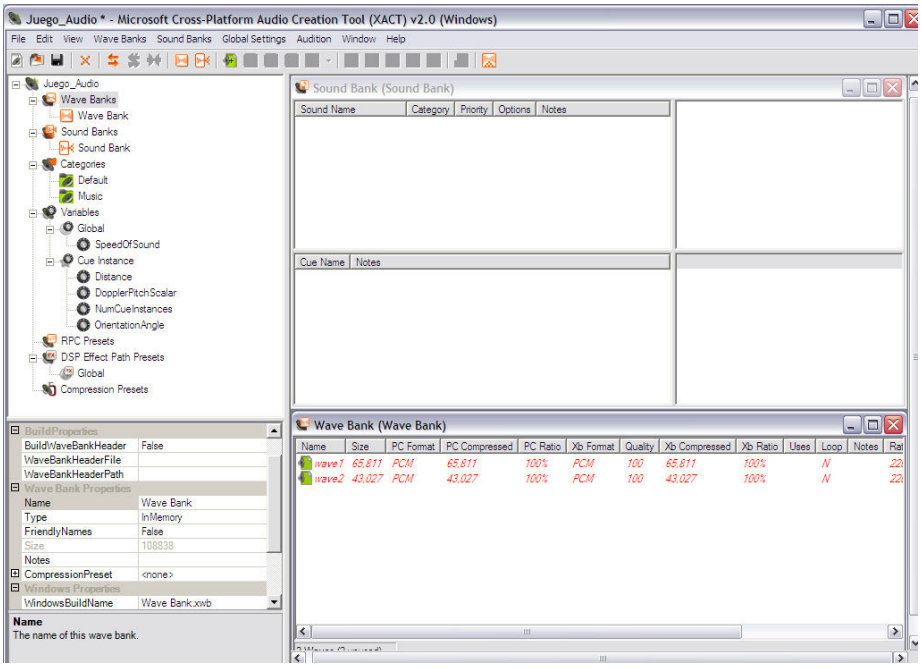
Ya en la aplicación seguiremos los siguientes pasos:

- Crea un nuevo proyecto ***File → New Project***
- En este momento, salva el proyecto con el nombre que desees, que se localizará en la carpeta ***Content/Audio***
- Ahora crea el *Wave Bank,* esto lo logras posicionándote sobre ***Wave Banks;*** con botón derecho da clic y aparecerá la opción de ***New Wave Bank ,*** como se muestra en la figura
- Haz el mismo procedimiento con Sound Bank
- Después de crear los dos bancos habrás creado 2 ventanas como se muestra en la imagen.

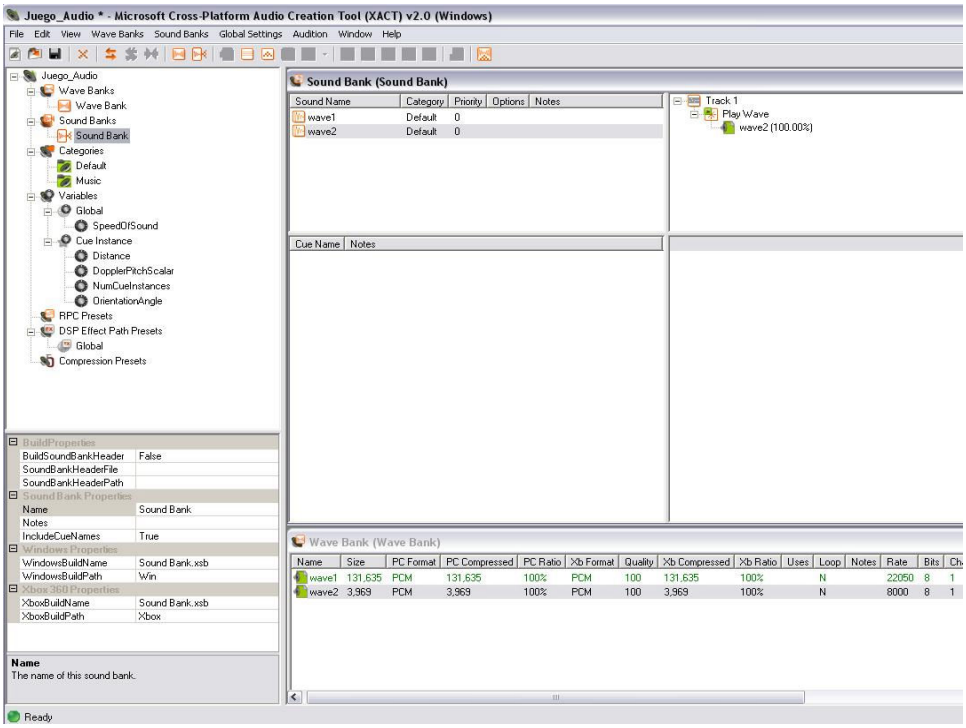




- Ahora es el momento de cargar los archivos .wav contenidos en el CD anexo dentro de ***Wave Bank***; para ello, te colocas sobre la ventana correspondiente y con el botón derecho del mouse das click, te aparecerá un menú donde tienes que elegir ***Insert Wave File(s)***.
Así en el directorio eliges los archivos .wav, en este caso *wave1.wav* y *wave2.wav* como se muestra en la figura.

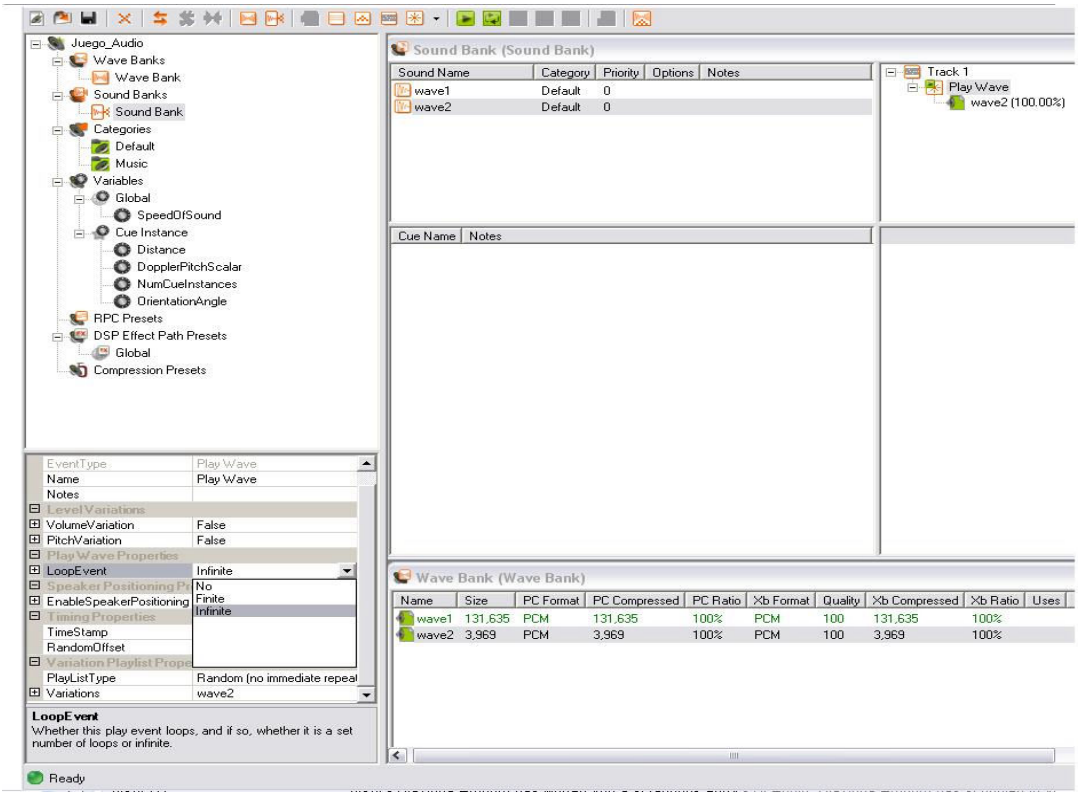


- Jala cada uno de los archivos de audio que están en el ***Wave Bank*** hacia la ventana ***Sound Bank*** y a su vez inclúyelos en la ventana ***Cue Name***



- Una vez que están cargados al dar doble clic sobre uno de ellos, sobre el *Sound Bank* aparecerá del lado derecho el *Play Wave,* que al seleccionarlo nos desplegará todas las propiedades del archivo sobre la ventana inferior izquierda.

- Modificaremos la opción de ***Loop Event*** a ***infinite***, de acuerdo a lo que queramos, en esta opción *infinite* lo reproduce de manera constante hasta que le indiquemos lo contrario. Para activar esta posibilidad basta abrir la pestaña y se despliega la lista, elegir *infnite.*



- Salva el proyecto.

Ahora regresa de nuevo al Microsoft Visual C# 2005. Si tu proyecto no está cargado aún, sigue estos pasos:

- Sobre la carpeta ***Content/Audio*** con el botón derecho del Mouse da clic
- Selecciona ***Agregar→Elemento existente***; selecciona nuestro proyecto, en este caso llamado ***Juego_Audio.xap***

Está ahora todo listo para agregar nuestro código, pondremos en color verde las líneas que añadimos.

Sobre el método ***Initialize*** agrega lo siguiente:

```
AudioEngine audioEngine;
WaveBank waveBank;
SoundBank soundBank;

protected override void Initialize()
{
   audioEngine = new AudioEngine( "Content\\Audio\\ Juego_Audio.xgs" );
   waveBank = new WaveBank( audioEngine, "Content\\Audio\\Wave Bank.xwb" );
   soundBank = new SoundBank( audioEngine, "Content\\Audio\\Sound Bank.xsb" );
   base.Initialize();
}
```

En la tabla anterior tenemos la conexión con el XACT, la que hace referencia al *engine* de Audio, así como con los *Wave* y *Sound Bank*.
Posteriormente, sobre el método *Update* incluiremos 2 lineas para actualizar lo referente al audio

```
protected override void Update( GameTime gameTime )
{
   if (GamePad.GetState( PlayerIndex.One ).Buttons.Back == ButtonState.Pressed)
      this.Exit();

   // Get some input.
   UpdateInput();

   // Update audioEngine.
   audioEngine.Update();
```

Finalmente, sobre el método ***Input*** indicaremos el evento en el que se asociará cada uno de los archivos de audio

```
protected void UpdateInput()
{
   // Get the game pad state.
   GamePadState currentState = GamePad.GetState( PlayerIndex.One );
   if (currentState.IsConnected)
   {
      // Rotate the model using the left thumbstick, and scale it down.
      modelRotation -= currentState.ThumbSticks.Left.X * 0.10f;

      // Create some velocity if the right trigger is down.
      Vector3 modelVelocityAdd = Vector3.Zero;
      // Find out what direction we should be thrusting, using rotation.
      modelVelocityAdd.X = -(float)Math.Sin( modelRotation );
      modelVelocityAdd.Z = -(float)Math.Cos( modelRotation );

      // Now scale our direction by how hard the trigger is down.
      modelVelocityAdd *= currentState.Triggers.Right;

      // Finally, add this vector to our velocity.
      modelVelocity += modelVelocityAdd;
```

```
        GamePad.SetVibration( PlayerIndex.One, currentState.Triggers.Right,
            currentState.Triggers.Right );

        // Set some audio based on whether we're pressing a trigger.
        if (currentState.Triggers.Right > 0)
        {
            if (engineSound == null)
            {
                engineSound = soundBank.GetCue( "wave1" );
                engineSound.Play();
            }

            else if (engineSound.IsPaused)
            {
                engineSound.Resume();
            }
        }
        else
        {
            if (engineSound != null && engineSound.IsPlaying)
            {
                engineSound.Pause();
            }
        }
        // In case you get lost, press A to warp back to the center.
        if (currentState.Buttons.A == ButtonState.Pressed)
        {
            modelPosition = Vector3.Zero;
            modelVelocity = Vector3.Zero;
            modelRotation = 0.0f;
            // Make a sound when we warp.
            soundBank.PlayCue( "wave2" );
        }
    }
}
```

Aquí está el código completo:

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion

public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    ContentManager content;


    public Game1()
    {
```

```
    graphics = new GraphicsDeviceManager( this );
    content = new ContentManager( Services );
}

AudioEngine audioEngine;
WaveBank waveBank;
SoundBank soundBank;

protected override void Initialize()
{
    audioEngine = new AudioEngine( "Content\\Audio\\ Juego_Audio.xgs" );
    waveBank = new WaveBank( audioEngine, "Content\\Audio\\Wave Bank.xwb" );
    soundBank = new SoundBank( audioEngine, "Content\\Audio\\Sound Bank.xsb" );
    base.Initialize();
}

// Specify the 3D model to draw.
Model myModel;

// The aspect ratio determines how to scale 3d to 2d projection.
float aspectRatio;

protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        myModel = content.Load<Model>("Content\\Models\\p1_wedge");
    }

    aspectRatio = graphics.GraphicsDevice.Viewport.Width /
        graphics.GraphicsDevice.Viewport.Height;
}

protected override void UnloadGraphicsContent( bool unloadAllContent )
{
    if (unloadAllContent == true)
    {
        content.Unload();
    }
}


// Set the velocity of the model, applied each frame to the model's position.
Vector3 modelVelocity = Vector3.Zero;



protected override void Update( GameTime gameTime )
{
    if (GamePad.GetState( PlayerIndex.One ).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Get some input.
    UpdateInput();

    // Update audioEngine.
    audioEngine.Update();

    // Add velocity to the current position.
    modelPosition += modelVelocity;

    // Bleed off velocity over time.
    modelVelocity *= 0.95f;
```

```
    base.Update( gameTime );
}

// Cue so we can hang on to the sound of the engine.
Cue engineSound = null;

protected void UpdateInput()
{
    // Get the game pad state.
    GamePadState currentState = GamePad.GetState( PlayerIndex.One );
    if (currentState.IsConnected)
    {
        // Rotate the model using the left thumbstick, and scale it down.
        modelRotation -= currentState.ThumbSticks.Left.X * 0.10f;

        // Create some velocity if the right trigger is down.
        Vector3 modelVelocityAdd = Vector3.Zero;

        // Find out what direction we should be thrusting, using rotation.
        modelVelocityAdd.X = -(float)Math.Sin( modelRotation );
        modelVelocityAdd.Z = -(float)Math.Cos( modelRotation );

        // Now scale our direction by how hard the trigger is down.
        modelVelocityAdd *= currentState.Triggers.Right;

        // Finally, add this vector to our velocity.
        modelVelocity += modelVelocityAdd;

        GamePad.SetVibration( PlayerIndex.One, currentState.Triggers.Right,
            currentState.Triggers.Right );

        // Set some audio based on whether we're pressing a trigger.
        if (currentState.Triggers.Right > 0)
        {
            if (engineSound == null)
            {
                engineSound = soundBank.GetCue( "wave1" );
                engineSound.Play();
            }

            else if (engineSound.IsPaused)
            {
                engineSound.Resume();
            }
        }
        else
        {
            if (engineSound != null && engineSound.IsPlaying)
            {
                engineSound.Pause();
            }
        }

        // In case you get lost, press A to warp back to the center.
        if (currentState.Buttons.A == ButtonState.Pressed)
        {
            modelPosition = Vector3.Zero;
            modelVelocity = Vector3.Zero;
            modelRotation = 0.0f;

            // Make a sound when we warp.
            soundBank.PlayCue( "wave2" );
```

```
            }
        }
    }
    // Set the position of the model in world space, and set the rotation.
    Vector3 modelPosition = Vector3.Zero;
    float modelRotation = 0.0f;

    // Set the position of the camera in world space, for our view matrix.
    Vector3 cameraPosition = new Vector3( 0.0f, 50.0f, -5000.0f );

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

        // Copy any parent transforms.
        Matrix[] transforms = new Matrix[myModel.Bones.Count];
        myModel.CopyAbsoluteBoneTransformsTo( transforms );

        // Draw the model. A model can have multiple meshes, so loop.
        foreach (ModelMesh mesh in myModel.Meshes)
        {
            // This is where the mesh orientation is set, as well as our camera and projection.
            foreach (BasicEffect effect in mesh.Effects)
            {
                effect.EnableDefaultLighting();
                effect.World = transforms[mesh.ParentBone.Index] * Matrix.CreateRotationY( modelRotation )
                    * Matrix.CreateTranslation( modelPosition );
                effect.View = Matrix.CreateLookAt( cameraPosition, Vector3.Zero, Vector3.Up );
                effect.Projection = Matrix.CreatePerspectiveFieldOfView( MathHelper.ToRadians( 45.0f ),
                    aspectRatio, 1.0f, 10000.0f );
            }
            // Draw the mesh, using the effects set above.
            mesh.Draw();
        }

        base.Draw( gameTime );
    }
}
```

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360
- Archivos de audio (.wav) *(CD anexo)*

Desarrollo

Laboratorio

- Reproduce este tutorial de nueva cuenta, utilizando nuevos archivos de audio y el modelo que has desarrollado
- Agrega 2 nuevos eventos de sonido al archivo *mini.fbx*, ya sea un claxon para el auto, frenos etc.
- Agrega música de fondo para la aplicación

Preguntas de Control

No hay preguntas de control para esta práctica

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.

Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. XNA Game Programming. Wiley Publishing. 2007
- ms-help://MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/GoingBeyond_Audio.htmPráctica

Práctica

6

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Sistema de Cámaras

Objetivos:

- Creación y manejo de cámaras
- Cámaras en primera persona
- Cámaras en tercera persona

Introducción

Para crear un sistema de cámaras en XNA es necesario incluir nuevos elementos que nos permitan movernos dentro de nuestro entorno. Entre los sistemas de cámara tenemos:

- **Cámara en primera persona**
  En este esté tipo de sistema la cámara actúa como sujeto de acción, es decir, como la cabeza de nuestro personaje y permite al espectador tener una sensación de total inmersión. Su rango de movimiento depende del sujeto a imitar. Muchos juegos de rol contienen esté tipo de cámara.

- **Cámara en tercera Persona**
  La cámara en tercera persona se concentra en visualizar nuestro sujeto o *avatar* y los movimientos y acciones en su entorno. Los juegos de aventura y RPG son los juegos más usuales donde usamos este sistema.

  Echemos un vistazo al código.

  Para crear movimientos de traslación y rotación de nuestra cámara agregaremos una matriz que, operada con los vectores, nos dará por resultado dicha trasformación.

- Declaramos las siguientes matrices: la primera se encargará de definir la posición de la cámara, mientras que la segunda se ocupa de los parámetros de la misma, como su abertura y alcance.

```
Matrix view;
Matrix proj;
```

- *Vector3* será nuestra referencia relativa a la rotación, y ayudará a definir la posición de nuestro *avatar* o nuestra representación virtual de la persona o jugador

```
// Avatar position and rotation variables.
Vector3 avatarPosition = new Vector3(0, 0, -20);
Vector3 avatarHeadOffset = new Vector3(0, 10, 0);
float avatarYaw = 0;
// The direction the camera points without rotation.
Vector3 cameraReference = new Vector3(0, 0, 10);
```

- A continuación definimos la velocidad de rotación, ángulo de visión, y la distancia entre los planos en la que podremos visualizar nuestros objetos.

```
// Rates in world units per 1/60th second (the default fixed step interval)
float rotationSpeed = 1f / 60f;
float forwardSpeed = 500f / 60f;
// Field of view of the camera in radians (pi/4 is 45 degrees).
static float viewAngle = MathHelper.PiOver4;
//Distance from the camera of the near and far clipping planes
static float nearClip = 5.0f;
static float farClip = 2000.0f;
// The camera state, avatar's center, first-person, third-person.
int cameraState = 0;
bool cameraStateKeyDown = false;
```

- Dentro del método ***Update***, además de los dispositivos que ya teníamos, agregaremos una actualización de nuestra posición del sujeto o jugador.

```
protected override void Update(GameTime gameTime)
{
   // Allows the default game to exit on Xbox 360 and Windows
   if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
      this.Exit();
       GetCurrentCamera();
   UpdateAvatarPosition();
   base.Update(gameTime);
}
```

- Definimos los tipos de cámara:

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

```
switch (cameraState)
{
    default:
    case 0:
        UpdateCamera();
        break;
    case 1:
        UpdateCameraFirstPerson();
        break;
    case 2:
        UpdateCameraThirdPerson();
        break;
}
```

y sus eventos:

```
// Update the position and direction of the avatar.
void UpdateAvatarPosition()
{
    KeyboardState keyboardState = Keyboard.GetState();
    GamePadState currentState = GamePad.GetState(PlayerIndex.One);
    if (keyboardState.IsKeyDown(Keys.Left) || (currentState.DPad.Left == ButtonState.Pressed))
    {
        // Rotate left.
        avatarYaw += rotationSpeed;
    }
    if (keyboardState.IsKeyDown(Keys.Right) || (currentState.DPad.Right == ButtonState.Pressed))
    {
        // Rotate right.
        avatarYaw -= rotationSpeed;
    }
    if (keyboardState.IsKeyDown(Keys.Up) || (currentState.DPad.Up == ButtonState.Pressed))
    {
        Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
```

```
            Vector3 v = new Vector3(0, 0, forwardSpeed);
            v = Vector3.Transform(v, forwardMovement);
            avatarPosition.Z += v.Z;
            avatarPosition.X += v.X;
        }
        if (keyboardState.IsKeyDown(Keys.Down) || (currentState.DPad.Down == ButtonState.Pressed))
        {
            Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
            Vector3 v = new Vector3(0, 0, -forwardSpeed);
            v = Vector3.Transform(v, forwardMovement);
            avatarPosition.Z += v.Z;
            avatarPosition.X += v.X;
        }
    }
    void GetCurrentCamera()
    {
        KeyboardState keyboardState = Keyboard.GetState();
        GamePadState currentState = GamePad.GetState(PlayerIndex.One);

        // Toggle the state of the camera.
        if (keyboardState.IsKeyDown(Keys.Tab) || (currentState.Buttons.LeftShoulder ==
ButtonState.Pressed))
        {
            cameraStateKeyDown = true;
        }
        else if (cameraStateKeyDown == true)
        {
            cameraStateKeyDown = false;
            cameraState += 1;
            cameraState %= 3;
        }
    }
    void UpdateCamera()
    {
```

```
    // Calculate the camera's current position.
    Vector3 cameraPosition = avatarPosition;
    Matrix rotationMatrix = Matrix.CreateRotationY(avatarYaw);
    // Create a vector pointing the direction the camera is facing.
    Vector3 transformedReference = Vector3.Transform(cameraReference, rotationMatrix);
    // Calculate the position the camera is looking at.
    Vector3 cameraLookat = cameraPosition + transformedReference;
    // Set up view matrix and projection matrix.
    view = Matrix.CreateLookAt(cameraPosition, cameraLookat, new Vector3(0.0f, 1.0f, 0.0f));
    Viewport viewport = graphics.GraphicsDevice.Viewport;
    float aspectRatio = (float)viewport.Width / (float)viewport.Height;
    proj = Matrix.CreatePerspectiveFieldOfView(viewAngle, aspectRatio, nearClip, farClip);
}
void UpdateCameraFirstPerson()
{
    Matrix rotationMatrix = Matrix.CreateRotationY(avatarYaw);



    // Transform the head offset so the camera is positioned properly relative to the avatar.
    Vector3 headOffset = Vector3.Transform(avatarHeadOffset, rotationMatrix);
    // Calculate the camera's current position.
    Vector3 cameraPosition = avatarPosition + headOffset;
    // Create a vector pointing the direction the camera is facing.
    Vector3 transformedReference = Vector3.Transform(cameraReference, rotationMatrix);
    // Calculate the position the camera is looking at.
    Vector3 cameraLookat = transformedReference + cameraPosition;
    // Set up view matrix and projection matrix
    view = Matrix.CreateLookAt(cameraPosition, cameraLookat, new Vector3(0.0f, 1.0f, 0.0f));
    Viewport viewport = graphics.GraphicsDevice.Viewport;
    float aspectRatio = (float)viewport.Width / (float)viewport.Height;
    proj = Matrix.CreatePerspectiveFieldOfView(viewAngle, aspectRatio, nearClip, farClip);
}
void UpdateCameraThirdPerson()
{
```

```
        Matrix rotationMatrix = Matrix.CreateRotationY(avatarYaw);
        // Create a vector pointing the direction the camera is facing.
        Vector3 transformedReference = Vector3.Transform(thirdPersonReference, rotationMatrix);
        // Calculate the position the camera is looking from.
        Vector3 cameraPosition = transformedReference + avatarPosition;
        // Set up view matrix and projection matrix
        view = Matrix.CreateLookAt(cameraPosition, avatarPosition, new Vector3(0.0f, 1.0f, 0.0f));
        Viewport viewport = graphics.GraphicsDevice.Viewport;
        float aspectRatio = (float)viewport.Width / (float)viewport.Height;
        proj = Matrix.CreatePerspectiveFieldOfView(viewAngle, aspectRatio, nearClip, farClip);
    }
```

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360

Desarrollo

Laboratorio

- Crea un sistema de cámaras que contengan 1era y 3era persona y que puedan intercambiarse de una a otra a través de una tecla.
- Crea una cámara que siga la trayectoria de una curva

Preguntas de Control

1. ¿ Cual es la diferencia en un sistema de cámaras entre 1 era y 3 era persona, en términos de programación?

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.

Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. **XNA Game Programming**. Wiley Publishing, 2007.
- ms-help://MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/Math_HowTo_ScriptedCamera.htm

Práctica

7

NOMBRE DEL ALUMNO:	
INSTRUCTOR:	
FECHA:	
FASE:	
VIDEOJUEGOS CON XNA	

Skybox

Objetivos:

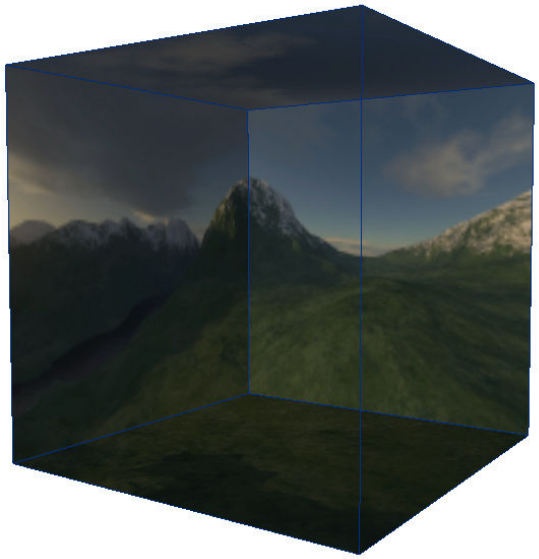
- Preparación de imágenes
- Creación de Skybox

Introducción

El proceso de gráficas en 3D, así como las aplicaciones en tiempo real, requieren una importante optimización de recursos. Skybox es una forma de crear fondos o *backgrounds,* que ayudan a crear un entorno tridimensional y lo hacen parecer aún más grande de lo que es, evitando construir grandes escenarios que pudieran representar una cantidad importante de polígonos. De igual manera sirve para delimitar nuestro espacio de juego.

Skybox consiste en un cubo con 6 texturas adheridas en cada una de sus caras con ángulos de visión de 90 grados de diferencia.

Para obtener estas texturas existen muchas fuentes, una de ellas es obtener las diferentes vistas a partir de un modelador simplemente rotando la cámara y obteniendo el *render* de cada una de ellas. Sin embargo, en la actualidad se pueden obtener a partir de *softwares* que generan diversos escenarios, como Bryce, Terragen, etc.
Se recomienda, para mejores resultados, obtener imágenes a una resolución no menor de 1024x1024.



Para generar un Skybox en XNA necesitamos 6 texturas como éstas:

Front.jpg	Right.jpg	Back.jpg	Left.jpg	Top.jpg	Bottom.jpg

Texturas obtenidas del curso Programación con Direct X Game Institute http://www.gameinstitute.com/

Michael Morton nos muestra como hacer un Skybox
Si deseas consultarlo se encuentra en:
http://www.ziggyware.com/readarticle.php?article_id=71

```
region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion

namespace SkyBox
{

   class SkyBox : GameComponent,IDrawable
   {

      Texture2D[] textures = new Texture2D[6];
      Effect effect;

      VertexBuffer vertices;
      IndexBuffer indices;
      VertexDeclaration vertexDecl;

      Vector3 vCameraDirection;
      Vector3 vCameraPosition;

      Matrix viewMatrix;
      Matrix projectionMatrix;
      Matrix worldMatrix;


      ContentManager content;

      public SkyBox(Game g)
         : base(g)
      {
      }

      public Vector3 CameraDirection
      {
         get { return vCameraDirection; }
```

```
    set { vCameraDirection = value; }
}

public Vector3 CameraPosition
{
    get { return vCameraPosition; }
    set
    {
        vCameraPosition = value;
        worldMatrix = Matrix.CreateTranslation(vCameraPosition);
    }
}

public Matrix ViewMatrix
{
    set { viewMatrix = value; }
    get { return viewMatrix; }
}

public Matrix ProjectionMatrix
{
    set { projectionMatrix = value; }
    get { return projectionMatrix; }
}

public ContentManager ContentManager
{
    set { content = value; }
}

public override void Initialize()
{
    base.Initialize();

    textures[0] = content.Load<Texture2D>("Skybox\\back");
    textures[1] = content.Load<Texture2D>("Skybox\\front");
    textures[2] = content.Load<Texture2D>("Skybox\\bottom");
    textures[3] = content.Load<Texture2D>("Skybox\\top");
    textures[4] = content.Load<Texture2D>("Skybox\\left");
    textures[5] = content.Load<Texture2D>("Skybox\\right");

    effect = content.Load<Effect>("Skybox\\skybox");

    IGraphicsDeviceService graphicsService = (IGraphicsDeviceService)
        Game.Services.GetService(typeof(IGraphicsDeviceService));

    vertexDecl = new VertexDeclaration(graphicsService.GraphicsDevice,
        new VertexElement[] {
            new VertexElement(0,0,VertexElementFormat.Vector3,
                VertexElementMethod.Default,
                VertexElementUsage.Position,0),
            new VertexElement(0,sizeof(float)*3,VertexElementFormat.Vector2,
                VertexElementMethod.Default,
                VertexElementUsage.TextureCoordinate,0)});


    vertices = new VertexBuffer(graphicsService.GraphicsDevice,
                    typeof(VertexPositionTexture),
                    4 * 6,
                    ResourceUsage.WriteOnly);

    VertexPositionTexture[] data = new VertexPositionTexture[4*6];
```

```
Vector3 vExtents = new Vector3(2500, 2500, 2500);
//back
data[0].Position = new Vector3(vExtents.X, -vExtents.Y, -vExtents.Z);
data[0].TextureCoordinate.X = 1.0f; data[0].TextureCoordinate.Y = 1.0f;
data[1].Position = new Vector3(vExtents.X, vExtents.Y, -vExtents.Z);
data[1].TextureCoordinate.X = 1.0f; data[1].TextureCoordinate.Y = 0.0f;
data[2].Position = new Vector3(-vExtents.X, vExtents.Y, -vExtents.Z);
data[2].TextureCoordinate.X = 0.0f; data[2].TextureCoordinate.Y = 0.0f;
data[3].Position = new Vector3(-vExtents.X, -vExtents.Y, -vExtents.Z);
data[3].TextureCoordinate.X = 0.0f; data[3].TextureCoordinate.Y = 1.0f;

//front
data[4].Position = new Vector3(-vExtents.X, -vExtents.Y, vExtents.Z);
data[4].TextureCoordinate.X = 1.0f; data[4].TextureCoordinate.Y = 1.0f;
data[5].Position = new Vector3(-vExtents.X, vExtents.Y, vExtents.Z);
data[5].TextureCoordinate.X = 1.0f; data[5].TextureCoordinate.Y = 0.0f;
data[6].Position = new Vector3(vExtents.X, vExtents.Y, vExtents.Z);
data[6].TextureCoordinate.X = 0.0f; data[6].TextureCoordinate.Y = 0.0f;
data[7].Position = new Vector3(vExtents.X, -vExtents.Y, vExtents.Z);
data[7].TextureCoordinate.X = 0.0f; data[7].TextureCoordinate.Y = 1.0f;

//bottom
data[8].Position = new Vector3(-vExtents.X, -vExtents.Y, -vExtents.Z);
data[8].TextureCoordinate.X = 1.0f; data[8].TextureCoordinate.Y = 0.0f;
data[9].Position = new Vector3(-vExtents.X, -vExtents.Y, vExtents.Z);
data[9].TextureCoordinate.X = 1.0f; data[9].TextureCoordinate.Y = 1.0f;
data[10].Position = new Vector3(vExtents.X, -vExtents.Y, vExtents.Z);
data[10].TextureCoordinate.X = 0.0f; data[10].TextureCoordinate.Y = 1.0f;
data[11].Position = new Vector3(vExtents.X, -vExtents.Y, -vExtents.Z);
data[11].TextureCoordinate.X = 0.0f; data[11].TextureCoordinate.Y = 0.0f;

//top
data[12].Position = new Vector3(vExtents.X, vExtents.Y, -vExtents.Z);
data[12].TextureCoordinate.X = 0.0f; data[12].TextureCoordinate.Y = 0.0f;
data[13].Position = new Vector3(vExtents.X, vExtents.Y, vExtents.Z);
data[13].TextureCoordinate.X = 0.0f; data[13].TextureCoordinate.Y = 1.0f;
data[14].Position = new Vector3(-vExtents.X, vExtents.Y, vExtents.Z);
data[14].TextureCoordinate.X = 1.0f; data[14].TextureCoordinate.Y = 1.0f;
data[15].Position = new Vector3(-vExtents.X, vExtents.Y, -vExtents.Z);
data[15].TextureCoordinate.X = 1.0f; data[15].TextureCoordinate.Y = 0.0f;


//left
data[16].Position = new Vector3(-vExtents.X, vExtents.Y, -vExtents.Z);
data[16].TextureCoordinate.X = 1.0f; data[16].TextureCoordinate.Y = 0.0f;
data[17].Position = new Vector3(-vExtents.X, vExtents.Y, vExtents.Z);
data[17].TextureCoordinate.X = 0.0f; data[17].TextureCoordinate.Y = 0.0f;
data[18].Position = new Vector3(-vExtents.X, -vExtents.Y, vExtents.Z);
data[18].TextureCoordinate.X = 0.0f; data[18].TextureCoordinate.Y = 1.0f;
data[19].Position = new Vector3(-vExtents.X, -vExtents.Y, -vExtents.Z);
data[19].TextureCoordinate.X = 1.0f; data[19].TextureCoordinate.Y = 1.0f;

//right
data[20].Position = new Vector3(vExtents.X, -vExtents.Y, -vExtents.Z);
data[20].TextureCoordinate.X = 0.0f; data[20].TextureCoordinate.Y = 1.0f;
data[21].Position = new Vector3(vExtents.X, -vExtents.Y, vExtents.Z);
data[21].TextureCoordinate.X = 1.0f; data[21].TextureCoordinate.Y = 1.0f;
data[22].Position = new Vector3(vExtents.X, vExtents.Y, vExtents.Z);
data[22].TextureCoordinate.X = 1.0f; data[22].TextureCoordinate.Y = 0.0f;
data[23].Position = new Vector3(vExtents.X, vExtents.Y, -vExtents.Z);
data[23].TextureCoordinate.X = 0.0f; data[23].TextureCoordinate.Y = 0.0f;

vertices.SetData<VertexPositionTexture>(data);
```

```
    indices = new IndexBuffer(graphicsService.GraphicsDevice,
                    typeof(short),6*6,
                    ResourceUsage.WriteOnly);

    short[] ib = new short[6 * 6];

    for (int x = 0; x < 6; x++)
    {
        ib[x * 6 + 0] = (short) (x * 4 + 0);
        ib[x * 6 + 2] = (short) (x * 4 + 1);
        ib[x * 6 + 1] = (short) (x * 4 + 2);

        ib[x * 6 + 3] = (short) (x * 4 + 2);
        ib[x * 6 + 5] = (short) (x * 4 + 3);
        ib[x * 6 + 4] = (short) (x * 4 + 0);
    }

    indices.SetData<short>(ib);

}

public override void Update(GameTime gameTime)
{

    base.Update(gameTime);
}


#region IDrawable Members

public void Draw(GameTime gameTime)
{
    if (vertices == null)
        return;

    effect.Begin();
    effect.Parameters["worldViewProjection"].SetValue(
                    worldMatrix * viewMatrix * projectionMatrix);


    for (int x = 0; x < 6; x++)
    {
        float f=0;
        switch(x)
        {
        case 0: //back
            f = Vector3.Dot(vCameraDirection,new Vector3(0,0,1));
            break;
        case 1: //front
            f = Vector3.Dot(vCameraDirection,new Vector3(0,0,-1));
            break;
        case 2: //bottom
            f = Vector3.Dot(vCameraDirection,new Vector3(0,1,0));
            break;
        case 3: //top
            f = Vector3.Dot(vCameraDirection,new Vector3(0,-1,0));
            break;
        case 4: //left
            f = Vector3.Dot(vCameraDirection,new Vector3(1,0,0));
            break;
        case 5: //right
```

```
                f = Vector3.Dot(vCameraDirection,new Vector3(-1,0,0));
                break;
            }

            if (f <= 0)
            {
                IGraphicsDeviceService graphicsService = (IGraphicsDeviceService)
                    Game.Services.GetService(typeof(IGraphicsDeviceService));

                GraphicsDevice device = graphicsService.GraphicsDevice;
                device.VertexDeclaration = vertexDecl;
                device.Vertices[0].SetSource(vertices, 0,
                    vertexDecl.GetVertexStrideSize(0));

                device.Indices = indices;

                effect.Parameters["baseTexture"].SetValue(textures[x]);
                effect.Techniques[0].Passes[0].Begin();


                device.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                    0,x*4,4,x*6,2);
                effect.Techniques[0].Passes[0].End();

            }
        }

        effect.End();
    }

    public int DrawOrder
    {
        get { return 0; }
    }

    public event EventHandler DrawOrderChanged;

    public bool Visible
    {
        get { return true; }
    }

    public event EventHandler VisibleChanged;

    #endregion
}

public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    ContentManager content;

    Model testMesh;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        content = new ContentManager(Services);
    }


    protected override void Initialize()
    {
```

```
    // TODO: Add your initialization logic here

    base.Initialize();
}


protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        SkyBox sb = new SkyBox(this);

        sb.ContentManager = this.content;

        sb.CameraDirection = new Vector3(0, 0, 1);
        sb.CameraPosition = new Vector3(0, 0, 0);

        sb.ViewMatrix = Matrix.CreateLookAt(sb.CameraPosition,
            sb.CameraPosition + sb.CameraDirection, new Vector3(0, 1, 0));

        Viewport viewport = graphics.GraphicsDevice.Viewport;
        float aspectRatio = (float)viewport.Width / (float)viewport.Height;

        sb.ProjectionMatrix = Matrix.CreatePerspectiveFieldOfView(
            MathHelper.PiOver4, aspectRatio, 1.0f, 10000.0f);



        sb.Initialize();


        this.Components.Add(sb);

        testMesh = content.Load<Model>("Meshes\\tank");
    }

    // TODO: Load any ResourceManagementMode.Manual content
}


protected override void UnloadGraphicsContent(bool unloadAllContent)
{
    if (unloadAllContent == true)
    {
        content.Unload();
    }
}

protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    SkyBox sb = (SkyBox)this.Components[0];

    Matrix matX = Matrix.CreateRotationY(
        -GamePad.GetState(PlayerIndex.One).ThumbSticks.Left.X * .05f);

    sb.CameraDirection = Vector3.TransformNormal(sb.CameraDirection, matX);

    Matrix matY = Matrix.CreateFromAxisAngle(
        Vector3.Cross(sb.CameraDirection,new Vector3(0,1,0)),
```

```
            -GamePad.GetState(PlayerIndex.One).ThumbSticks.Left.Y * .05f);

        Vector3 vDir = Vector3.TransformNormal(sb.CameraDirection, matY);

        if(Math.Abs(Vector3.Dot(new Vector3(0,1,0),vDir)) > 0.9f)
            vDir = sb.CameraDirection;
        sb.CameraDirection = vDir;

        sb.CameraPosition = sb.CameraPosition + sb.CameraDirection * (float)gameTime.TotalRealTime.TotalSeconds *
GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y;

        sb.CameraPosition = sb.CameraPosition + Vector3.Cross(new Vector3(0,1,0),sb.CameraDirection) *
(float)gameTime.TotalRealTime.TotalSeconds * -GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X;

        sb.ViewMatrix = Matrix.CreateLookAt(sb.CameraPosition,
            sb.CameraPosition + sb.CameraDirection, new Vector3(0, 1, 0));

        base.Update(gameTime);
    }

    private void DrawModel(Model m,Matrix view,Matrix projection)
    {
        Matrix[] transforms = new Matrix[m.Bones.Count];
        m.CopyAbsoluteBoneTransformsTo(transforms);


        foreach (ModelMesh mesh in m.Meshes)
        {
            foreach (BasicEffect effect in mesh.Effects)
            {
                effect.EnableDefaultLighting();

                effect.View = view;
                effect.Projection = projection;
                effect.World = mesh.ParentBone.Transform;
            }
            mesh.Draw();
        }
    }


    /// <summary>
    /// This is called when the game should draw itself.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
    protected override void Draw(GameTime gameTime)
    {
        graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

        SkyBox sb = (SkyBox)this.Components[0];
        DrawModel(testMesh,sb.ViewMatrix,sb.ProjectionMatrix);

        base.Draw(gameTime);
    }
  }
}
```

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360
- Modelador o software para generar Texturas de Skybox

Desarrollo

Laboratorio
• Crea con el software de tu preferencia una serie de imágenes para Skybox. • Cárgalos y logra una aplicación donde puedas navegar en el mundo.

Preguntas de Control

¿Qué otras formas de ambiente puedes implementar en tu juego?

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- http://www.ziggyware.com/readarticle.php?article_id=71

Práctica

8

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Colisiones

Objetivos:

- Introducción a Detección de Colisiones
- Bounding Volume
- Non-Bounding Volume

Introducción

Hemos desarrollado hasta el momento la capacidad de navegar en nuestro mundo virtual, de manipular cámaras, agregar sonido, entre otras cosas. Sin embargo, para crear una mejor experiencia para el usuario tenemos que implementar la presencia de simulaciones, leyes físicas y matemáticas en general.

Parte importante de estas aplicaciones es la detección de colisiones, que no es más que el proceso en el que dos o más objetos se sobreponen o coinciden en el mismo lugar en el espacio, y su interacción.

El método que seguiremos dentro de esta práctica será el de delimitar cada uno de los objetos por medio de una caja o esfera y éstos se aproximarán al volumen con el que haremos colisión.

Las posibilidades que tenemos dentro de las clases *Bounding Volume* o limite de volumen son las siguientes:

Bounding Volume

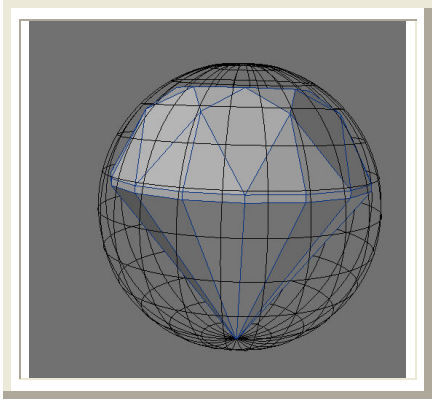
- ***Bounding Sphere***

Esta estructura es la determinada por una esfera.

Sintaxis:

```
[TypeConverterAttribute("typeof(Microsoft.Xna.Framework.Design.BoundingSphereConverter)")]
[SerializableAttribute]
public struct BoundingSphere : IEquatable<BoundingSphere>
```

Toma en cuenta sólo el centro y el radio de la esfera, lo cual lo hace una manera muy simple de detección, es decir, si tenemos 2 esferas, se encargará de calcular la suma de los radios y al comparar si el resultado es menor que la de ambos radios entonces se intersectan.



➢ Bounding Box

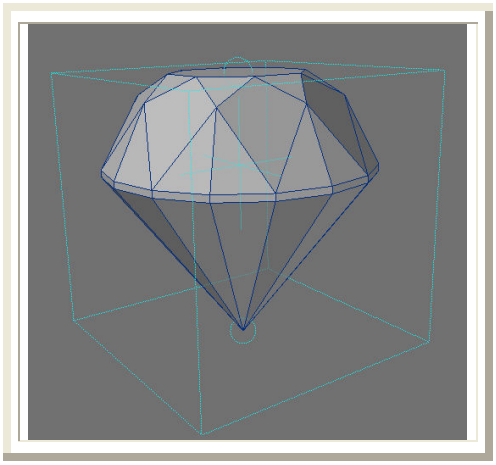
Esta estructura es la determinada por una caja.

Sintaxis:

```
[TypeConverterAttribute("typeof(Microsoft.Xna.Framework.Design.BoundingBoxConverter)")]
[SerializableAttribute]

public struct BoundingBox : IEquatable<BoundingBox>
```

Esta clase despliega una caja alrededor del objeto y se alinea de acuerdo al eje.
Suele utilizarse para hacer colisiones con edificios u objetos de tipo rectangular.

Una de las desventajas es que cuando rotamos nuestro objeto ya no estaría alineada a los ejes y por lo tanto tendríamos que recrear nuestra caja.



➢ Bounding Frustum

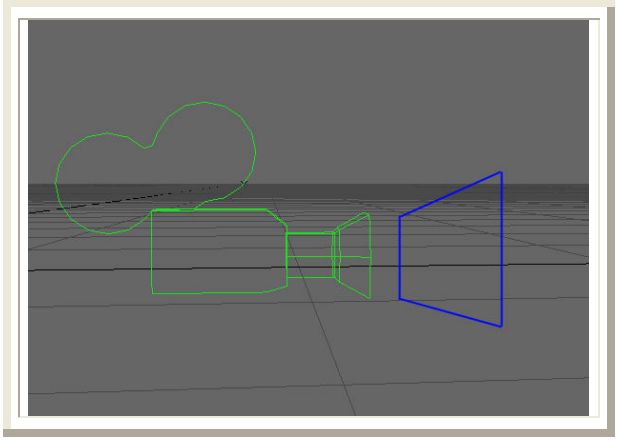
Definimos un frustum como un volumen en forma de pirámide entre planos paralelos y es justo lo que podemos observar a través de una cámara, como se observa en la ilustración siguiente.

Cuando hacemos el producto entre view matrix y projection matrix obtenemos el *Bounding frustum* o el volumen frustum, al igual que la Bounding Box se necesita recrear cada vez que sufre una transformación.

Se utiliza para detectar cuando un objeto colisiona con la cámara

Sintaxis:

```
[TypeConverterAttribute(System.ComponentModel.ExpandableObjectConverter, System,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089)] [SerializableAttribute] public
class BoundingFrustum : IEquatable<BoundingFrustum>
```



Existen también clases que no dependen de un limite de volumen, estos pueden ser:

Non- Bounding Volume

➢ Plano

Esta estructura es la determinada por un plano y se define por un vector perpendicular a éste. Esta clase se puede utilizar junto con una clase de volumen e indicar si se intersectan.

➢ Ray

Determina un rayo a partir de un punto en el espacio, y checa las distancias para ver si se intersectan o no.

➢ Model

Esta clase calcula un Bounding Sphere para cualquier modelo de nuestra escena y la intesección funciona como en esta última.

➢ Se puede obtener un tipo de estructura que se apegue más a nuestras necesidades aunque es una implementación que el propio usuario deberá crear.

En el siguiente código que se puede obtener en la documentación de Microsoft XNA podemos ver de forma mas clara su funcionamiento.

- Definimos la estructura.
- Consiste en utilizar Bounding Sphere para nuestros objetos y verificar si estos colisionan. Necesitamos cargar un objeto llamado "sphere", el cual interactuará con otras esferas colisionando entre ellas. Si existe una colisión tendremos como resultado *true.*
- Tenemos las *"endspheres"* que son las esferas límite situadas en ambos extremos y que estarán fijas haciendo colisión con las esferas interiores

```
struct WorldObject
{
  public Vector3 position;
  public Vector3 velocity;
```

```
 public Model model;
 public Texture2D texture2D;
 public Vector3 lastPosition;
 public void MoveForward()
{
  lastPosition = position;
  position += velocity; }
  public void Backup()
  {
  position -= velocity;
  }
  public void ReverseVelocity()
  {
  velocity.X = -velocity.X;
  }
}
 WorldObject sphere1;
WorldObject sphere2;
WorldObject endSphere1;
WorldObject endSphere2;
```

- Indicamos las colisiones y sus eventos

```
   static void CheckForCollisions( ref WorldObject c1, ref WorldObject c2 )
        { for (int i = 0; i < c1.model.Meshes.Count; i++)
        {
            // Check whether the bounding boxes of the two cubes intersect.
BoundingSphere c1BoundingSphere = c1.model.Meshes[i].BoundingSphere;
c1BoundingSphere.Center += c1.position;
for (int j = 0; j < c2.model.Meshes.Count; j++)
  {
  BoundingSphere c2BoundingSphere = c2.model.Meshes[j].BoundingSphere;
  c2BoundingSphere.Center += c2.position;
  if (c1BoundingSphere.Intersects( c2BoundingSphere ))
  {
   c2.ReverseVelocity();
```

```
    c1.Backup();
    c1.ReverseVelocity();
    return; } } } }
```

- Añadimos sobre el método ***Draw***:

```
DrawModel( sphere1.model, Matrix.CreateTranslation( sphere1.position ), sphere1.texture2D );
DrawModel( sphere2.model, Matrix.CreateTranslation( sphere2.position ), sphere2.texture2D );
DrawModel( endSphere1.model, Matrix.CreateTranslation( endSphere1.position ), endSphere1.texture2D );
DrawModel( endSphere2.model, Matrix.CreateTranslation( endSphere2.position ), endSphere2.texture2D );
```

- Sobre método ***Update***:

```
protected override void Update( GameTime gameTime )
{
  base.Update( gameTime );
  UpdateAvatarPosition();
   sphere1.MoveForward();
   CheckForCollisions( ref sphere1, ref sphere2 );
   CheckForCollisions( ref sphere1, ref endSphere1 );
   CheckForCollisions( ref sphere1, ref endSphere2 );
    sphere2.MoveForward();
    CheckForCollisions( ref sphere2, ref sphere1 );
    CheckForCollisions( ref sphere2, ref endSphere1 );
    CheckForCollisions( ref sphere2, ref endSphere2 );
  }
```

- Cargamos nuestro modelo y textura:

```
protected override void LoadGraphicsContent( bool loadAllContent )
{
```

```
    base.LoadGraphicsContent( loadAllContent );
    if (loadAllContent)
   {
        sphere = contentManager.Load<Model>( "sphere" );
        sphereTexture = contentManager.Load<Texture2D>( "spheretexture" );
        sphere1.model = sphere;
        sphere1.texture2D = sphereTexture;
        sphere2.model = sphere;
        sphere2.texture2D = sphereTexture;
         endSphere1.model = sphere;
         endSphere1.texture2D = sphereTexture;
         endSphere2.model = sphere;
         endSphere2.texture2D = sphereTexture;
         }
   }
```

- Establecemos la posición y velocidad de nuestros objetos, y fijamos las *"endspheres"*:

```
public Game1()
{
     graphics = new Microsoft.Xna.Framework.GraphicsDeviceManager( this );
     contentManager = new ContentManager( Services );
     sphere1 = new WorldObject();
     sphere1.position = new Vector3( 5, 0, 0 );
     sphere1.velocity = new Vector3( -0.1f, 0, 0 );
     sphere2 = new WorldObject();
     sphere2.position = new Vector3( -6, 0.5f, 0 );
     sphere2.velocity = new Vector3( 0.1f, 0, 0 );
     endSphere1 = new WorldObject();
     endSphere1.position = new Vector3( 10, 0, 0 );
    endSphere1.velocity = new Vector3( 0, 0, 0 );
    endSphere2 = new WorldObject();
    endSphere2.position = new Vector3( -10, 0, 0 );
    endSphere2.velocity = new Vector3( 0, 0, 0 );
}
```

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360
- 2 modelos (raqueta y bola) que se proporcionan para la práctica

Desarrollo

Laboratorio

- Reproduce este tutorial con el código aquí disponible y modificar los parámetros de colisión. Anota tus observaciones.
- Con los objetos llamados "**raqueta.fbx**" y "**bola.fbx**" haz un sistema de colisiones, donde simules un juego de ping pong con 2 raquetas y una bola. Los objetos han sido facilitados en el CD para su uso.
- Realiza un ejercicio similar al anterior pero en el que involucres los limites de volumen con los de no volumen.
- Con el tutorial del capitulo 3 modifícalo, haciendo de éste un escenario de un circuito de carreras, donde estará un auto, e implementa colisión con otros modelos como autos, edificios, barreras de contención etc.

Preguntas de Control

1. Describe para el caso del Bounding Box y el Bounding Sphere el modo en que colisionan con otro objeto.

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. **XNA Game Programming**. Wiley Publishing, 2007.
- ms-help://MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/Math_HowTo_DetectTwoObjectsColliding.htm
- http://msdn2.microsoft.com/en-us/library/bb313876.aspx
- http://www.ziggyware.com/readarticle.php?article_id=48

Práctica

9

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Interfaz de Usuario

Objetivos:

- Crear pantallas de usuario
- Reconocer la importancia de clases *Input* y *Update*
- Diseñar una interfaz

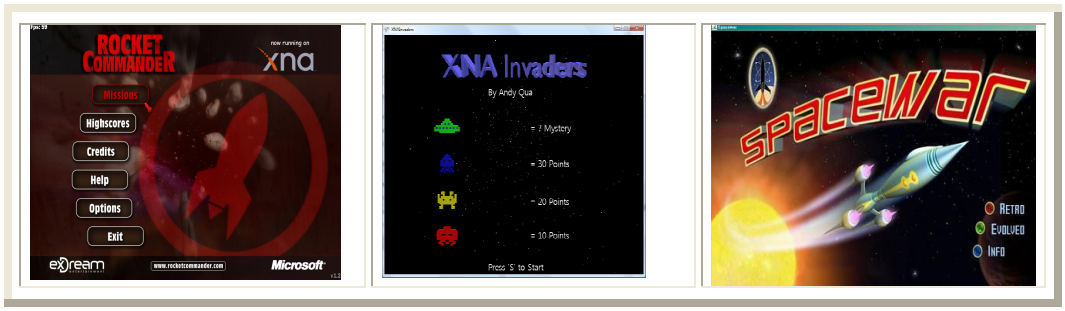
Introducción

Una interfaz es un medio por el cual el usuario puede interactuar con una aplicación. En este caso, podríamos decir que es la conexión entre el usuario y el juego, desde la cual podemos acceder a diversas pantallas y aplicaciones.

Entre las pantallas más comunes que debe contener un videojuego están:

- Menú principal
- Créditos
- Puntuación
- Opciones
- Ayuda
- Misiones
- Salida

Como es de suponer cada una de estas pantallas deberán tener no sólo botones que funcionen correctamente, sino estar acompañadas de un gráfico que pueda comunicar el contenido del juego y que cumpla con información y función estética.



Además de la interfaz gráfica de menú, existe otro tipo de interfaz que es el "skin o máscara". Está situado en el juego directamente como una plantilla y su función es la de informar sobre el status, progreso, o puntuación del juego.

La importancia de un buen gráfico es la de hacer que el usuario tenga una mejor experiencia e información.

Desde luego existen muchas formas de generar una interfaz de usuario, desde una muy simple con botón de inicio y salida hasta la más compleja con animaciones, fuentes y sonido, entre otras características.

Las herramientas varían desde flash, código XML, XAML, bibliotecas específicas para GUI, hasta utilizar solamente el XNA, todas ellas compatibles, añadiendo algunos controladores, códigos externos, etc.

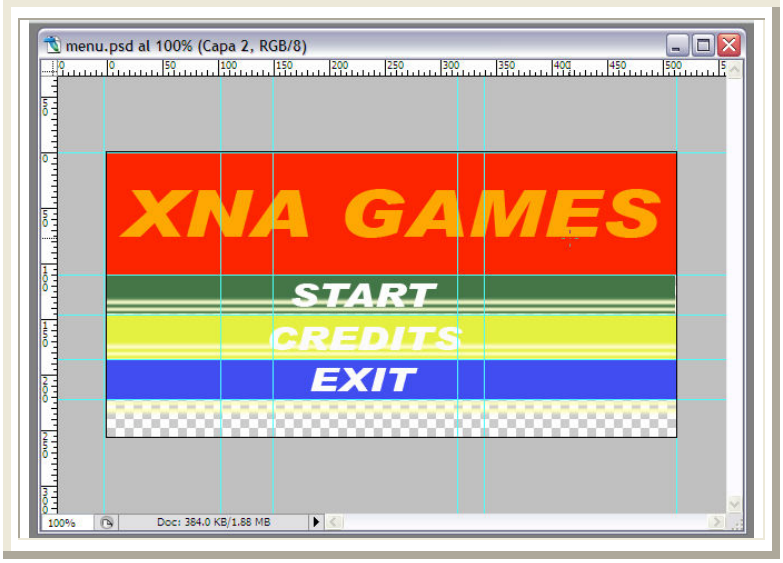
Dentro del código, la *clase input* es de gran importancia. Dentro de esta clase, daremos acceso a estados de teclado, mouse y gamepad, pero sin duda la que tiene mayor peso será *Update* , que es la que se encarga de supervisar la posición de nuestros dispositivos (mouse y gamepad) cuadro por cuadro.

Para ejemplificar esto, veremos una de las funciones sobre nuestro Menú de inicio.

Partiremos de un gráfico que contiene título, así como las entradas que consideramos necesarias.
Nota: tomaremos para este caso el fondo azul como independiente de las letras.

Si partimos el gráfico en rectángulos, delimitamos un área de acción donde podemos detectar la posición del mouse y posterior evento a realizar.

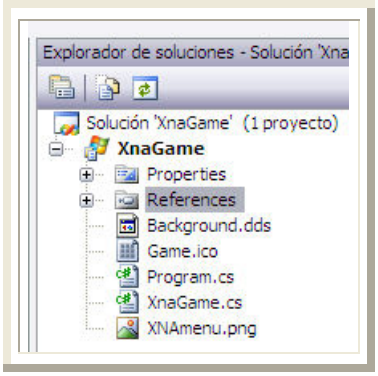
Nos queda algo como esto:



Los colores nos indican los límites de cada área medidos en pixeles, que podemos representar en código de la siguiente forma:

```
#region Constants

static readonly Rectangle
XnaGamesRect = new Rectangle(0, 0, 512, 110),
MenuStartRect = new Rectangle(0, 110, 512, 38),
MenuCreditsRect = new Rectangle(0, 148, 512, 38),
MenuExitRect = new Rectangle(0, 185, 512, 38),
```

Recordaremos cargar cada una de las texturas que utilizaremos en nuestro explorador de soluciones



El siguiente paso es declarar nuestras variables:

```
#region Variables
        GraphicsDeviceManager graphics;
        ContentManager content;
        Texture2D backgroundTexture, menuTexture,
gameTexture;
```

Cargando nuestro Content con texturas:

```
protected override void LoadGraphicsContent(bool loadAllContent)
        {
            if (loadAllContent)
            {
                // Create sprite batch
                spriteBatch = new
SpriteBatch(graphics.GraphicsDevice);

// Load all our content
backgroundTexture = content.Load<Texture2D>("Background");
menuTexture = content.Load<Texture2D>("XNAMenu");

} // if
base.LoadGraphicsContent(loadAllContent);
```

En el método *Update* definiremos los estados del gamepad: como dirección, abortar el juego y otros más. Este control sobre el dispositivo nos ayuda a la selección en el menú.

Sólo falta dibujar, esto lo logramos con la ayuda de spritebatch, de esta forma:

```
#region Sprite handling
        class SpriteToRender
        {
        public Texture2D texture;
        public Rectangle rect;
        public Rectangle? sourceRect;
        public Color color;
        public SpriteToRender(Texture2D setTexture, Rectangle setRect,
                                        Rectangle? setSourceRect, Color setColor)
        {
                        texture = setTexture;
                        rect = setRect;
                        sourceRect = setSourceRect;
                        color = setColor;
        } // SpriteToRender(setTexture, setRect, setColor)
        } // SpriteToRender

        List<SpriteToRender> sprites = new List<SpriteToRender>();
        SpriteBatch spriteBatch = null;

        public void RenderSprite(Texture2D texture, Rectangle rect, Rectangle? sourceRect,
        Color color)
        {
                        sprites.Add(new SpriteToRender(texture, rect, sourceRect, color));
        } // RenderSprite(texture, rect, sourceRect, color)

public void RenderSprite(Texture2D texture, Rectangle rect, Rectangle? sourceRect)
                {
                        RenderSprite(texture, rect, sourceRect, Color.White);
                } // RenderSprite(texture, rect, sourceRect)

public void RenderSprite(Texture2D texture, int x, int y, Rectangle? sourceRect,
                        Color color)
                {
                RenderSprite(texture,new Rectangle(x, y, sourceRect.Value.Width,
sourceRect.Value.Height),
                                        sourceRect, color);
                } // RenderSprite(texture, rect, sourceRect)

public void RenderSprite(Texture2D texture, int x, int y, Rectangle? sourceRect)
                {
                        RenderSprite(texture,new Rectangle(x, y, sourceRect.Value.Width,
sourceRect.Value.Height),
                                        sourceRect, Color.White);
                } // RenderSprite(texture, rect, sourceRect)

public void RenderSprite(Texture2D texture, Rectangle rect, Color color)
                {
                        RenderSprite(texture, rect, null, color);
                } // RenderSprite(texture, rect, color)

public void RenderSprite(Texture2D texture, Rectangle rect)
                {
                        RenderSprite(texture, rect, null, Color.White);
                } // RenderSprite(texture, rect)

public void RenderSprite(Texture2D texture)
                {
                        RenderSprite(texture, new Rectangle(0, 0, 1024, 768), null, Color.White);
                } // RenderSprite(texture)
```

```
public void DrawSprites()
                        {
                                // No need to render if we got no sprites this frame
                                if (sprites.Count == 0)
                                        return;

                                // Start rendering sprites
                                spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
                                        SpriteSortMode.BackToFront, SaveStateMode.None);

                                // Render all sprites
                                foreach (SpriteToRender sprite in sprites)
                                        spriteBatch.Draw(sprite.texture,
                                                // Rescale to fit resolution
                                                new Rectangle(
                                                sprite.rect.X * width / 1024,
                                                sprite.rect.Y * height / 768,
                                                sprite.rect.Width * width / 1024,
                                                sprite.rect.Height * height / 768),
                                                sprite.sourceRect, sprite.color);

                                // We are done, draw everything on screen with help of the end method.
                                spriteBatch.End();

                                // Kill list of remembered sprites
                                sprites.Clear();
                        } // DrawSprites()
                        #endregion
```

Limpiamos y cargamos el background:

```
protected override void Draw(GameTime gameTime)
                        {
                                // Clear background
                                graphics.GraphicsDevice.Clear(Color.Black);

                                // Draw background texture in a seperate pass, else it gets messed up with
                                // our other sprites, the ordering does not really work great.
                                spriteBatch.Begin();
                                spriteBatch.Draw(backgroundTexture,
                                        new Rectangle(0, 0, width, height),
                                        Color.LightGray);
                                spriteBatch.End();
```

Desplegamos los rectángulos dando funcionalidad a la selección de menú (Start y Exit), coloreando naranja el rectángulo seleccionado y en blanco en caso contrario.
Dentro de los parámetros de *RenderSprite* encontramos valores numéricos en pixeles que nos indican la posición de los rectángulos en pantalla.

```
// Show screen depending on our current screen mode
                    if (gameMode == GameMode.Menu)
                    {
                            // Show menu
                            RenderSprite(menuTexture,
                                    512-XnaGameRect.Width/2, 150, XnaGameRect);
                            RenderSprite(menuTexture,
                                    512-MenuStartRect.Width/2, 300, MenuStartRect,
                                    currentMenuItem == 0 ? Color.Orange : Color.White);
                            RenderSprite(menuTexture,
                                    512-MenuCreditsRect.Width/2, 350 MenuCreditsRect,
                                    currentMenuItem == 1 ? Color.Orange : Color.White);
                            RenderSprite(menuTexture,
                                    512-MenuExitRect.Width/2, 400, MenuExitRect,
                                    currentMenuItem == 2 ? Color.Orange : Color.White);
```

Nota: En el libro de **XNA Game Programming por Benjamín Nitschke** existe un capítulo especial para interfaz de usuario, en el cual está basado este código y que puedes consultar para mayor referencia.

Material

- Microsoft Visual Studio C# Express 2005
- XNA Game Studio Express
- Control Xbox 360
- Editor de Imagen

Desarrollo

Laboratorio

- Crea una interfaz Menú Inicio de Juego que contenga los siguientes campos con sus respectivas pantallas:
    1. Inicio
    2. Salir
    3. Puntuación
    4. Créditos
    5. Opciones

Preguntas de Control

1. Menciona a grandes rasgos el proceso para crear una interfaz de usuario de Inicio

Conclusiones

Escribe qué ventajas o desventajas encontraste en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- Nitschke Benjamín. **XNA Game Programming**. Wiley Publishing, 2007.
- http://msmvps.com/blogs/valentin/archive/2007/05/21/annexe-interface-utilisateur-en-xna.aspx
- http://forums.xna.com/thread/11972.aspx

Práctica

10

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Modelado de Personajes

Objetivos:

- Modelar un personaje en 3D

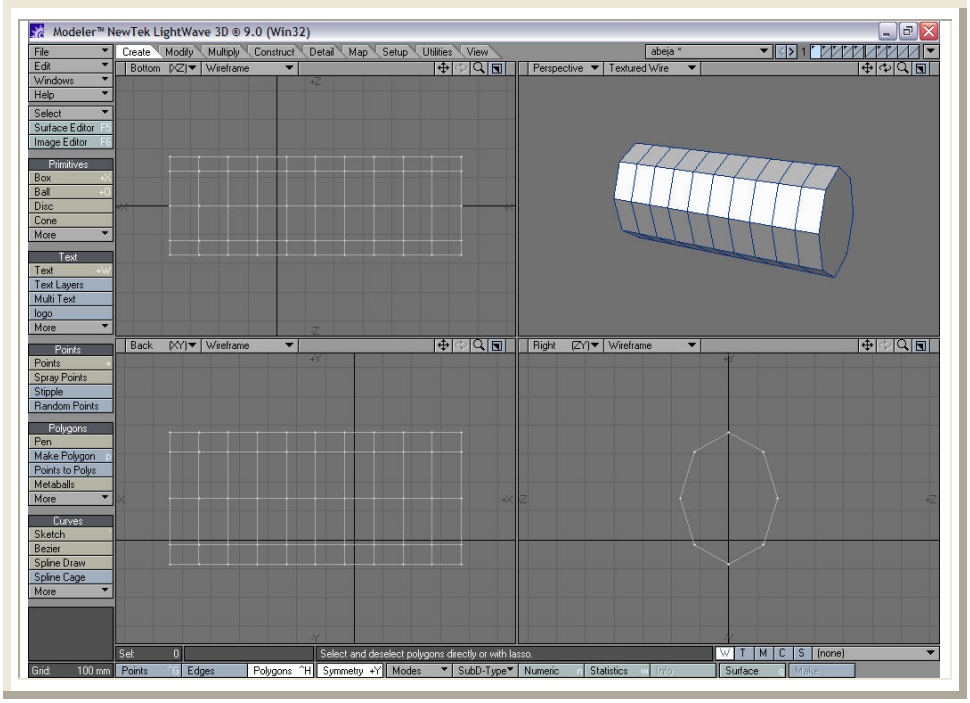
Introducción

En cuanto al modelado de personajes, recomendamos conseguir un par de dibujos que nos puedan servir como referencia.

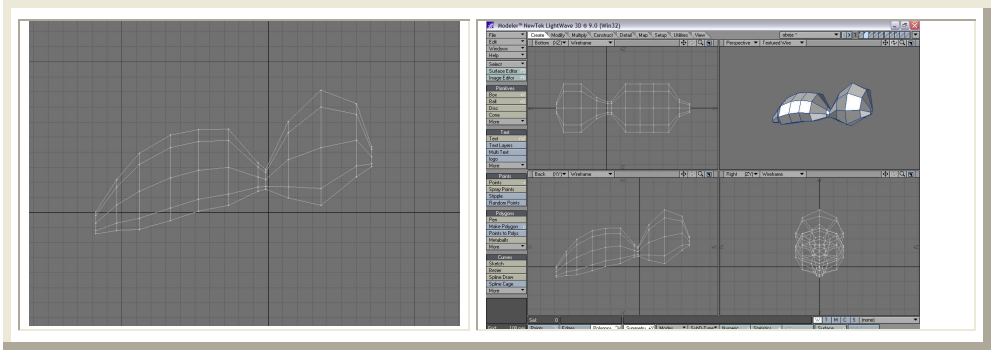
Aquí mostraremos a grandes rasgos el proceso a seguir para modelar una avispa, que forma parte del proyecto final en esta serie de prácticas.

Nuestro personaje lo modelaremos en lightwave 9, en el caso de tener referencias las cargamos abriendo ***display options*** **(d)** sobre la pestaña ***Backdrop.***

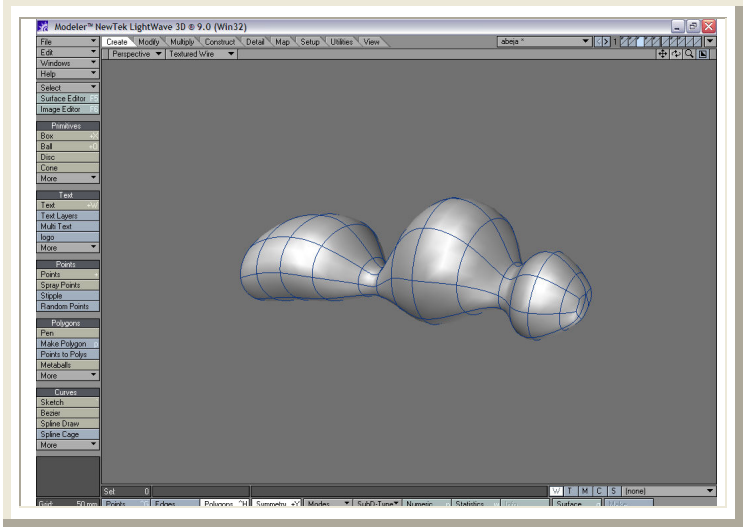
❖ Para crear el cuerpo de la avispa empezaremos con una *primitiva* llamada *disc*, que encontraremos en el menú situado del lado izquierdo de la pantalla, en la sección del mismo nombre. Damos profundidad y le ponemos algunas divisiones. Al oprimir la letra "n" aparecerá una ventana donde podremos especificar caracteristicas más detalladas sobre nuestra primitiva, ya sea número de divisiones, posición, tamaño, etc.



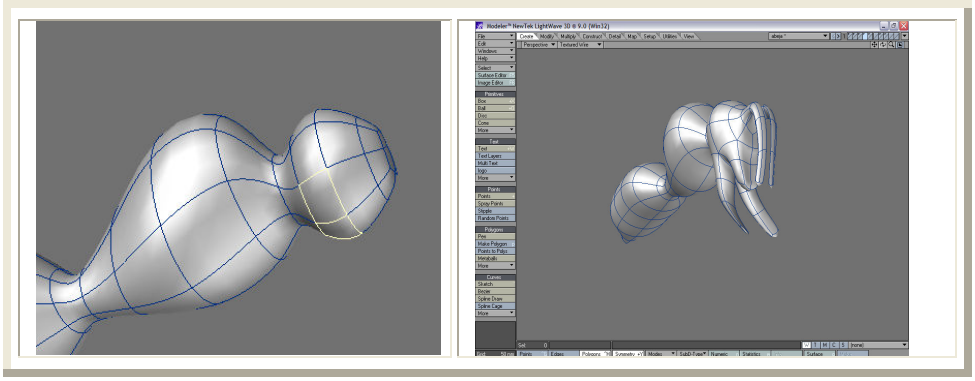
❖ El siguiente paso será seleccionar cada columna de puntos y aplicarles un stretch (h) en las diferentes vistas, con el que daremos forma al cuerpo de nuestra avispa; también moveremos los puntos para acomodarlos.



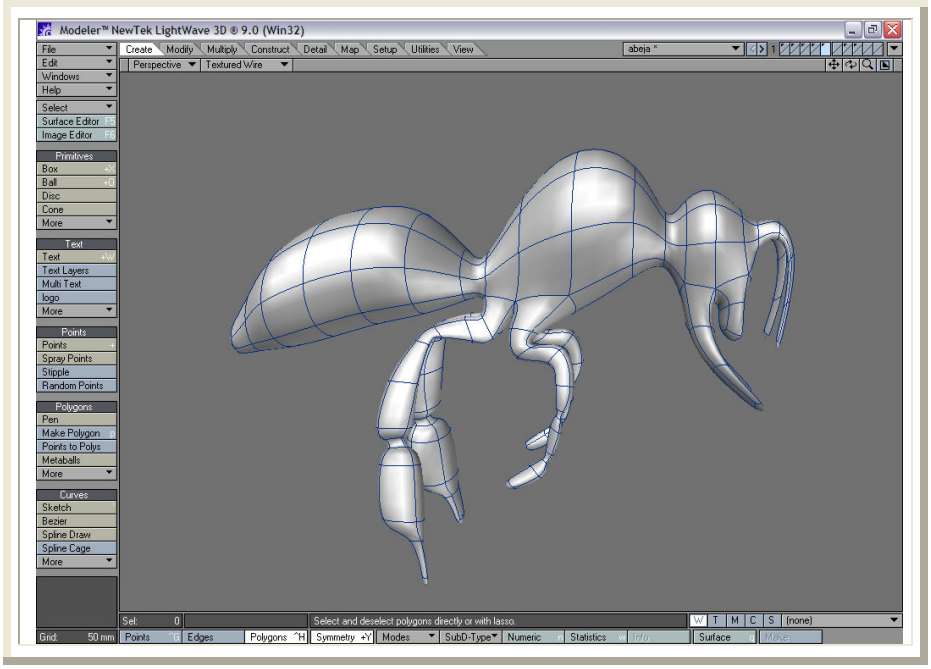
- Aplicamos un suavizado *subpatch (TAB)*



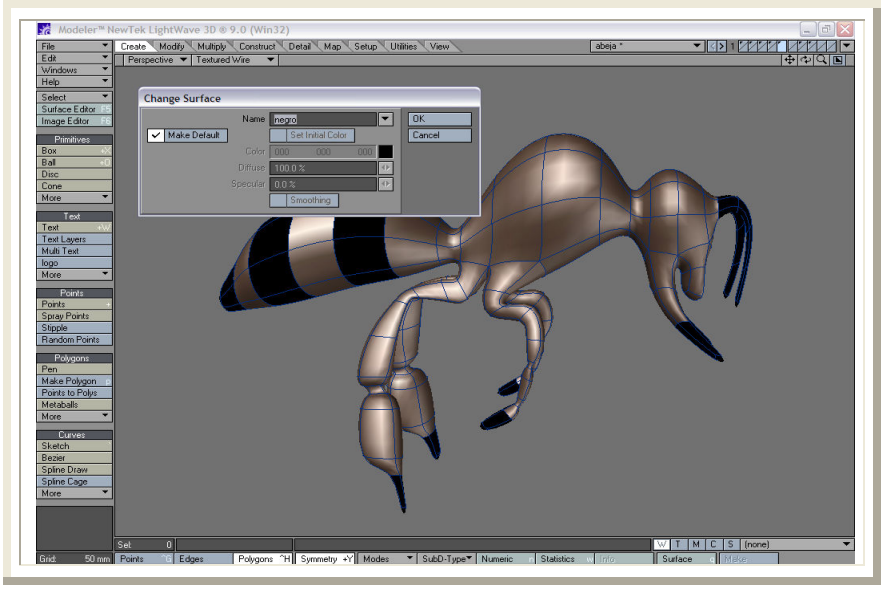
- Para las extremidades, tenemos que sacar geometría de nuestra primitiva. Para eso utilizamos una herramienta llamada bevel(b), la cual nos dará los polígonos que necesitamos. Este paso lo haremos las veces que sea necesario hasta completar las extremidades. Recuerda utilizar la letra "n" para mayor precisión.



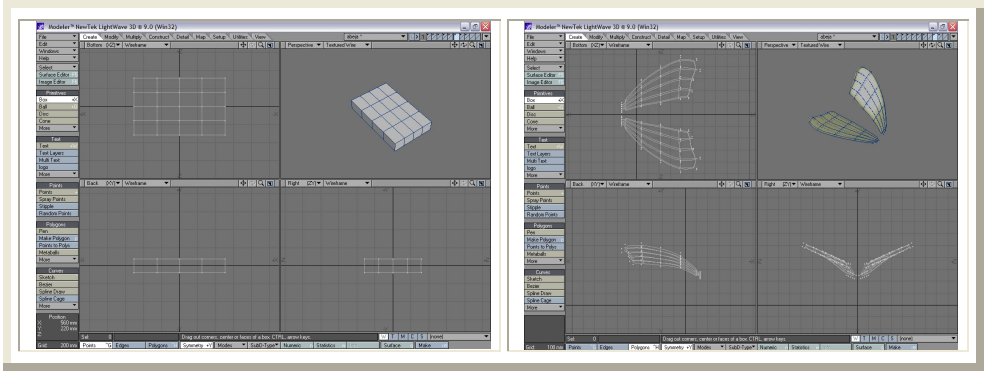
- Al aplicar varias veces el bevel sobre ambos polígonos nos quedará algo como esto:



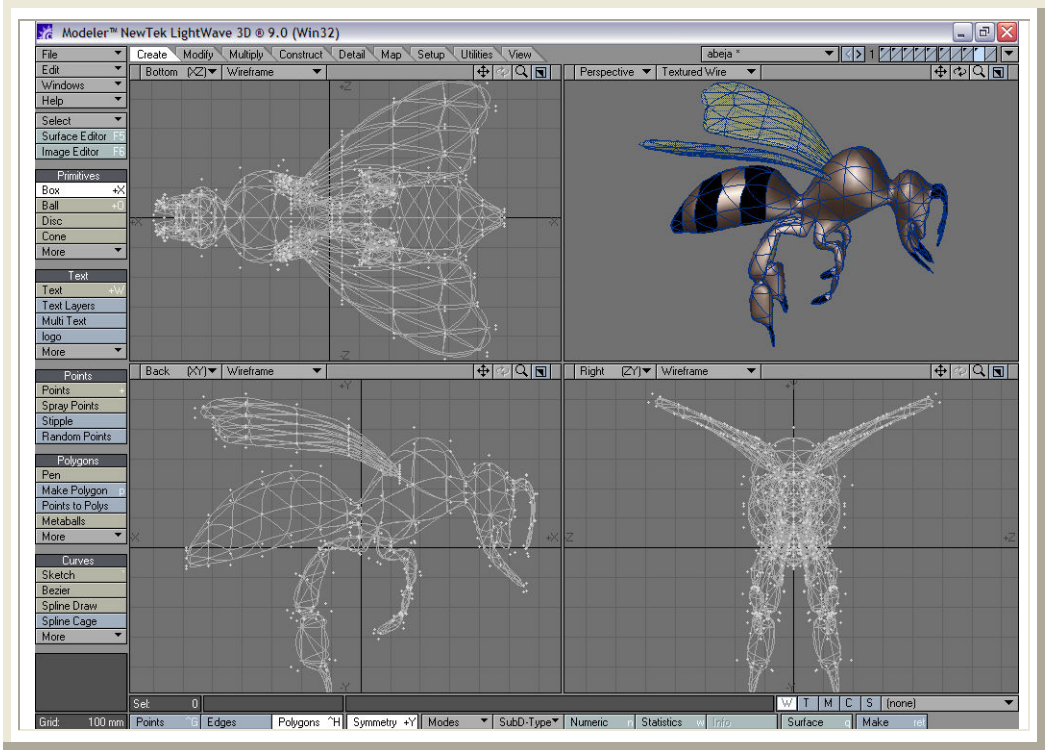
- Para definir texturas necesitamos seleccionar las áreas que nos interesan; esto se obtiene escogiendo con el mouse los polígonos que llevarán cierta superficie o textura, apretamos la letra "q" (change surface) y ponemos el nombre para identificarlo.
- Para modificar la textura bastará con apretar F5, y aparecerá el editor de superficies. Elige la que quieras modificar, ahí aparecerán todos los parámetros.



- La alas podemos obtenerlas a partir de un cubo; le ponemos algunas subdivisiones ("n") y con las herramientas *drag, move* y *stretch*, movemos cada punto hasta darle la forma de un ala. *Después aplicamos un mirror* para tener el par.



- El resultado final será algo como la siguiente imagen:



En el caso de las texturas los formatos de imagen que soporta XNA son:

.dds	.jpeg	.tga	.bmp
.png	.dxt		

Las texturas tendrán que ser aplicadas como UV map. Para mayor referencia consultar:

http://www.newtek.com/products/lightwave/tutorials/uvmapping/uvmapping/index.html

- Sólo resta salvar nuestro objeto en un formato compatible con XNA, éste puede ser .FBX o .X.

Si tu modelador no tiene esta opción puedes obtener el plugin en la página de Autodesk FBX.

http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6839916

Material

- Modelador 3D

Desarrollo

Laboratorio
❖ Diseña un personaje en 3 dimensiones.

Preguntas de Control

No hay preguntas de control para esta práctica.

Conclusiones

Describe las dificultades encontradas en el desarrollo de la práctica.
Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6839916
- http://67.15.36.49/team/Tutorials/benmathis/benmathis_1.asp
- http://members.shaw.ca/lightwavetutorials%20/characters.htm
- http://www.highend3d.com/maya/tutorials/

Práctica

11

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Animación con Huesos en XNA

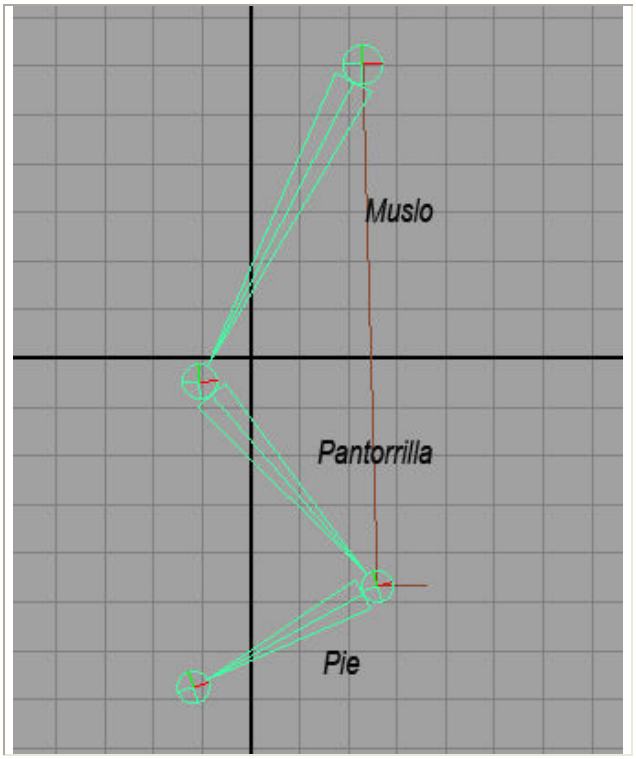
Objetivos:

- Introducción a huesos
- Crear Skeletal Animation
- Incorporar animación con huesos en XNA

Introducción

En esta sección abordaremos el tema de huesos en XNA, también llamada Skeletal Animation. Este tipo de animación requiere un conocimiento previo sobre animación de caracteres (*character animation).* Sin embargo, hay algunos sitios con tutoriales que explican paso a paso cómo conformar un sistema de huesos dentro del modelador. Sólo se abordará el tema de manera general y el alumno se encargará de generar su propio sistema.

Para generar una animación con huesos lo primero de que nos aseguraremos es de tener un buen modelo, tener la cantidad de geometría adecuada y su buena construcción hará que el movimiento no parezca "plástico" ni con deformaciones.
Un sistema de huesos está formado por un esqueleto, similar al humano en su estructura y funcionamiento, como en el ejemplo:



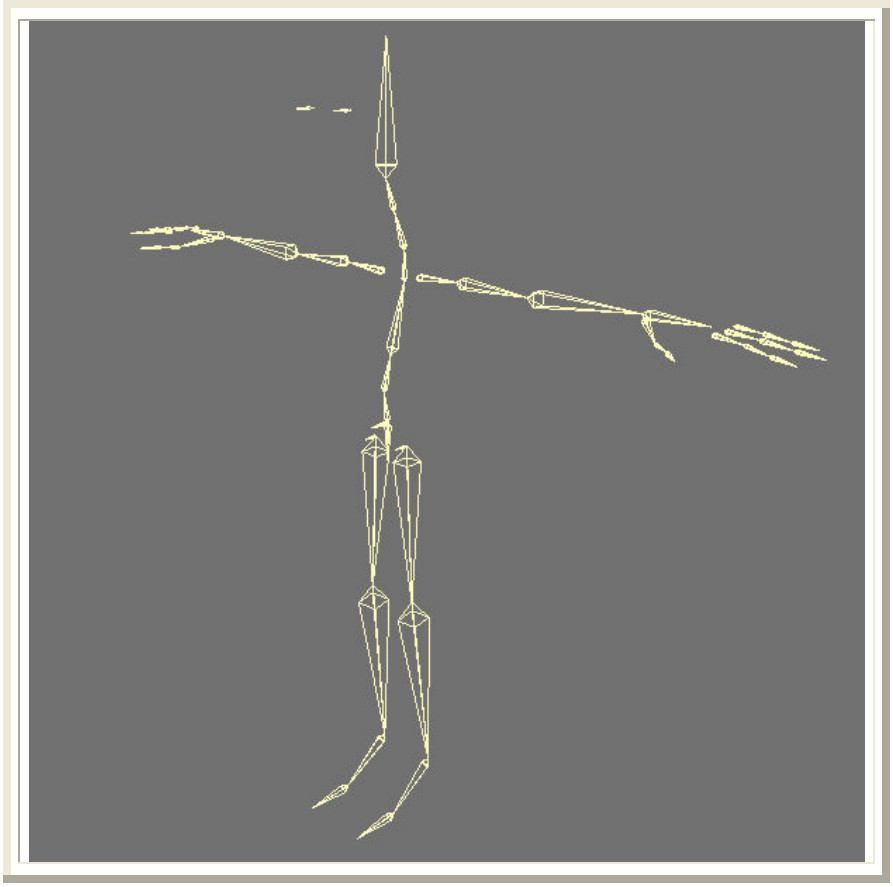
Una pierna puede estar formada por 3 huesos, el correspondiente al muslo, otro a la pantorrilla y finalmente al pie.

El hueso del pie dependerá del movimiento de la pantorrilla y éste a su vez del muslo, si este último se mueve también lo hará el resto de la pierna, a esto le llamamos *emparentamiento.*

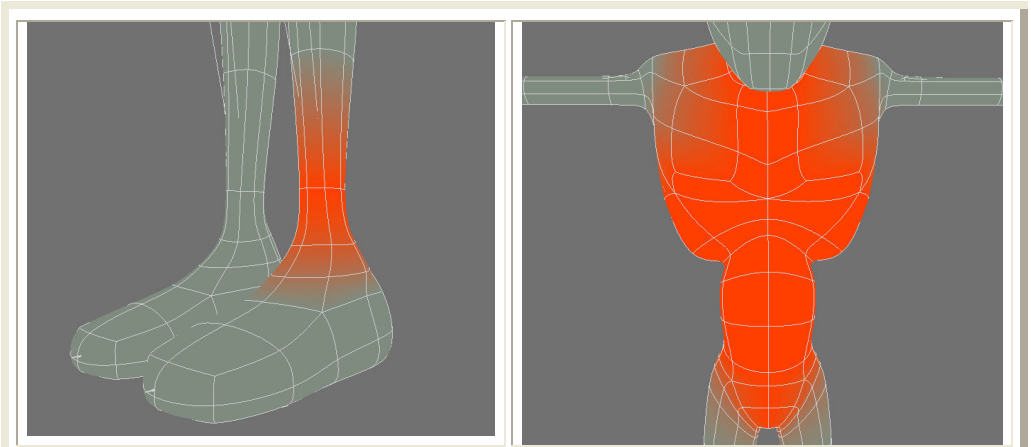
Pie →Pantorrilla
Pantorrilla →muslo

Llamaremos entonces a *muslo* "hueso raiz" o "hueso padre".

De la misma manera estará formado el resto del cuerpo, cada extremidad estará emparentada a la columna, donde estará un hueso base que mueve todo el cuerpo.



Junto con este sistema de huesos existen los mapas de peso, en ellos definiremos una mejor influencia sobre cada uno de ellos, haciendo un movimiento más natural. Se dará cierto peso para cada zona de la geometría y se asociará con el hueso que corresponda. Este proceso puede ser bastante laborioso pero es fundamental para lograr una buena animación.



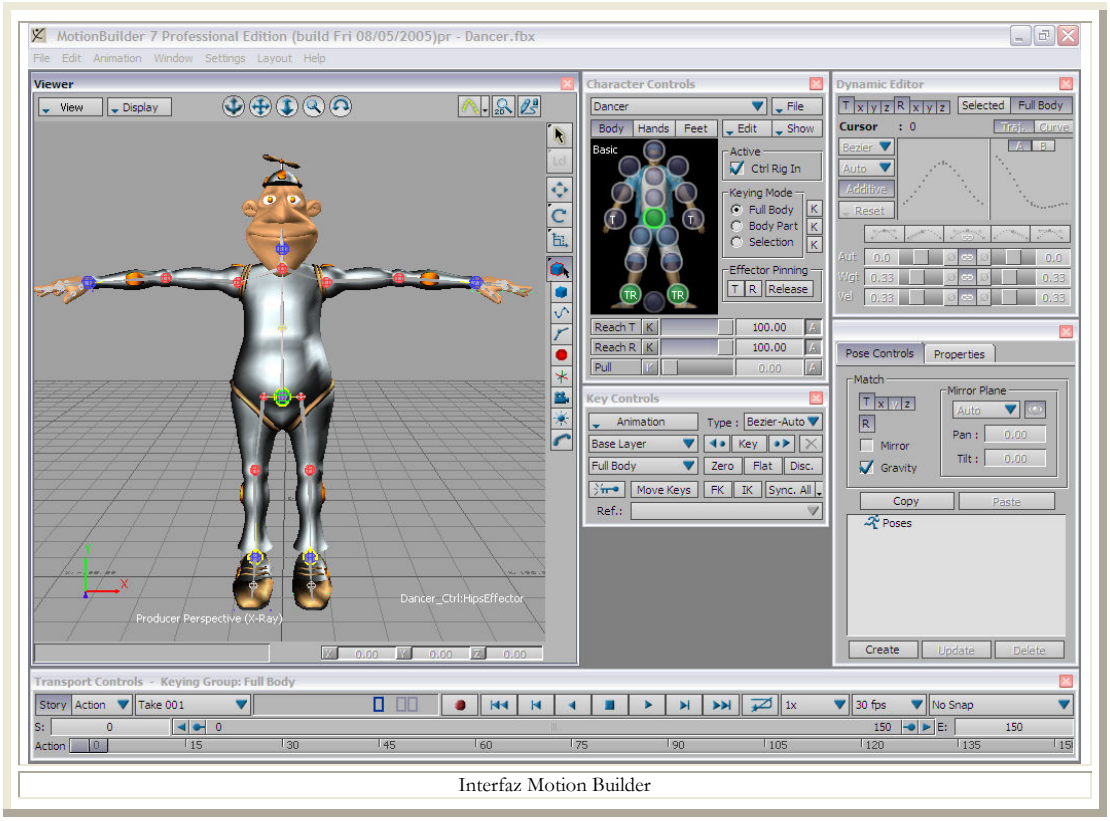
Peso en piernas | Peso en tronco

Una vez que tengamos el sistema de huesos del personaje (*setup),* el siguiente paso será animarlo. Este puede ser logrado a través del software antes mencionado utilizando llaves de animación.

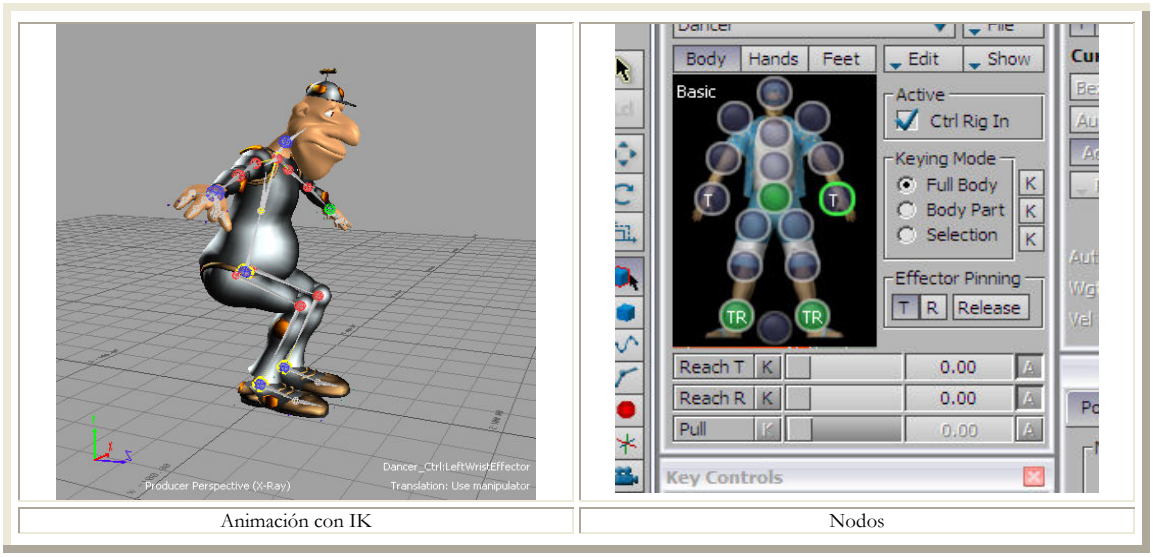
Una buena opción es utilizar ***Motion Builder***, un software que genera animación en base a bibliotecas de *motion capture,* la cual puede ser editada directamente en el programa y aplicada a nuestro personaje. Además genera un formato .FBX compatible con XNA como formato nativo. El único inconveniente es que ***Motion Builder*** no es un software gratuito. El contar con la licencia nos permitirá acortar los tiempos de producción y obtener excelentes resultados.

Cada hueso de nuestro sistema deberá ser nombrado de acuerdo al standard de ***Motion Builder***, de esta manera reconocerá cada uno de ellos y le asociará un *nodo* que contiene un sistema de Cinemática Inversa (IK), lo cual hará que nuestro personaje se mueva sin dificultad

Al igual que el software de animación en la parte inferior contiene un *Time Line* donde se pueden registrar las llaves de animación (Keys animation) que marcarán cada movimiento de nuestro personaje. La animación entre cada una de estas llaves se hará por interpolación.



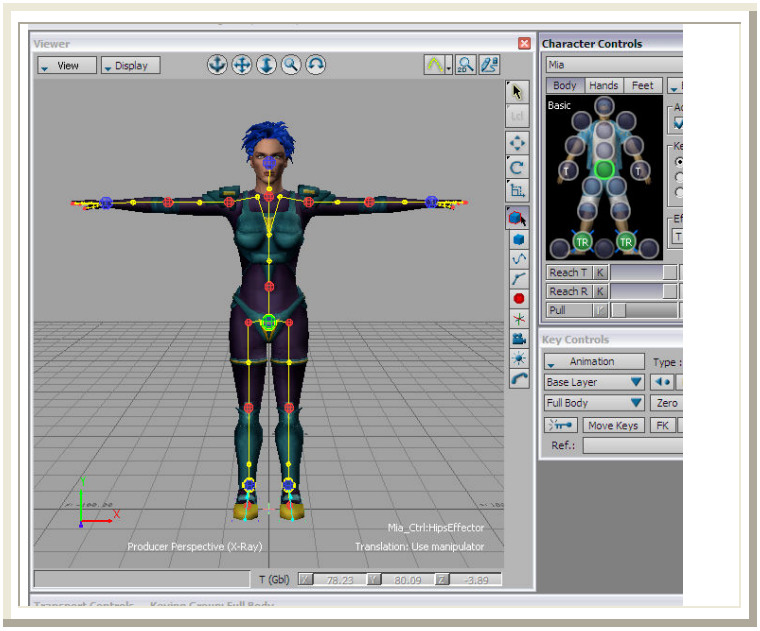
Interfaz Motion Builder



Animación con IK | Nodos

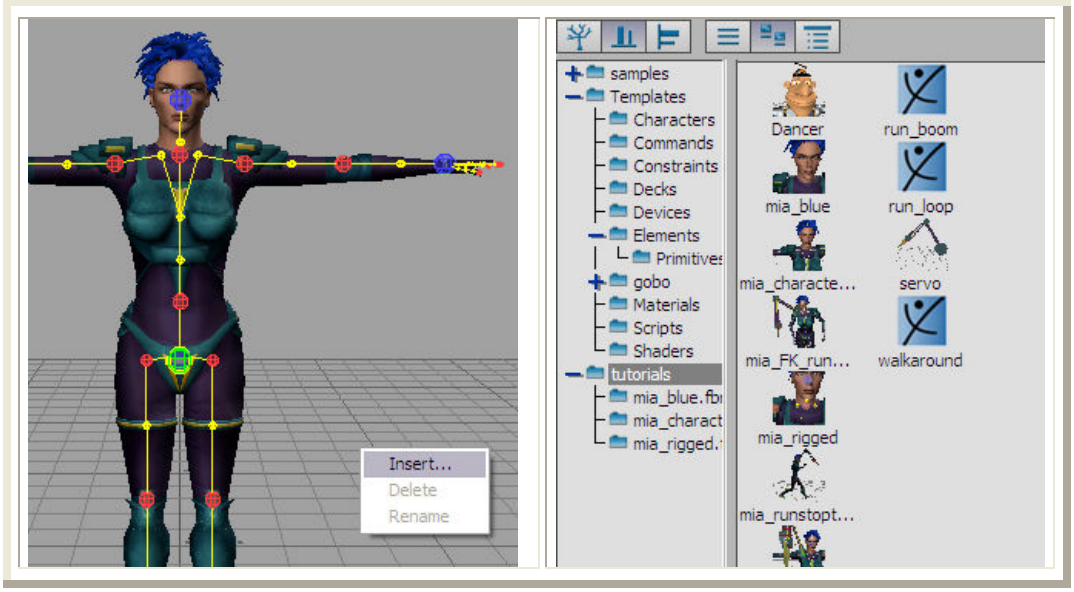
Las bibliotecas de movimiento son una excelente opción, sobre todo si el usuario tiene poca experiencia en animación de personajes. Los pasos para aplicar una biblioteca de movimiento son los siguientes:

1. Carga el modelo .FBX llamado *rigged*.
2. Da clic en cualquier unión de huesos, esto activará los nodos disponibles en la ventana *Characters Controls*

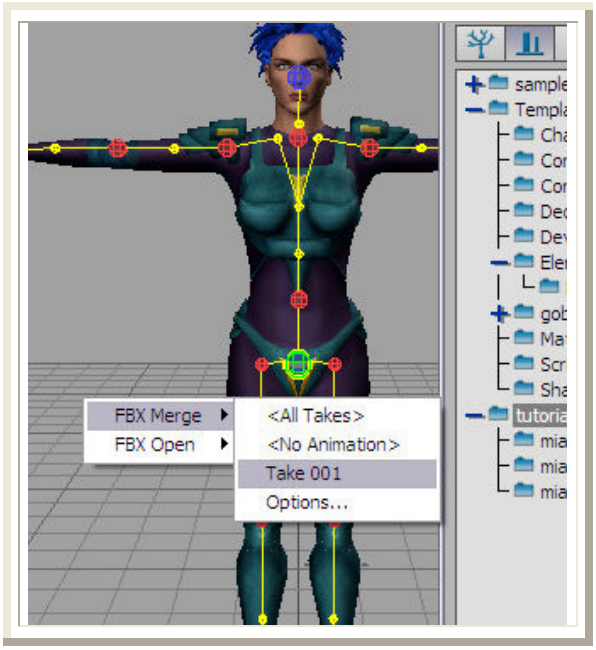


3. Para cargar un archivo de movimiento daremos click con botón derecho del mouse, abriéndose un menú donde seleccionamos **Insert**.

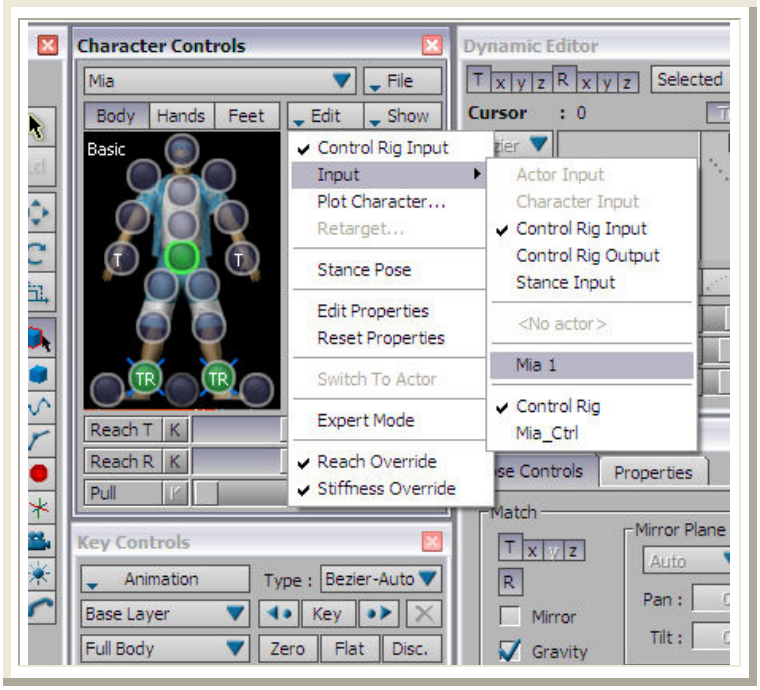
4. Aparecerá el *Asset Browser*, seleccionaremos una biblioteca y la arrastramos dentro de nuestra ventana *Viewer*., dando clic con el botón derecho.



5. Seleccionamos **Merge FBX → Take 001**



6. Ya tenemos nuestra animación junto con el modelo cargados, ahora iremos a nuestra ventana **Characters Controls**, seleccionamos la casilla **Edit → Input → Nombre del archivo.**



7. Da clic en **Play**.

De esta manera, haremos por separado todas las tomas o animaciones que necesitemos para nuestro juego.

En un videojuego las acciones más comunes son:

- Correr
- Caminar
- Detenerse
- Mover la cabeza

Entre otras.

Ahora pasemos al código.

Hace falta incorporar los huesos y animaciones dentro de XNA. Lo primero será declarar las variables de huesos y movimientos, en este caso el ejemplo sería para reconocer 3 de los huesos en XNA (Muslo, Pantorrilla y Pie).

```
//variables
        protected Model modelo;
        //movimiento
        protected float movimiento1 = 0.0f;
        protected float movimiento2 = 0.0f;
        protected float movimiento3 = 0.0f;
        //bones
        ModelBone bone_Muslo;
        ModelBone bone_Pantorrilla;
        ModelBone bone_Pie;
        //transformaciones iniciales
        Matrix transMuslo;
        Matrix transPantorrilla;
        Matrix transPie;
        //contenedor
        Matrix[] boneTransf;
```

- Cargamos los elementos

```
public void load(ContentManager content)
        {
            modelo = content.Load<Model>("personaje");
            boneMuslo = modelo.Bones["bone_Muslo"];
            bonePantorrilla = modelo.Bones["bone_Pantorrilla"];
            bonePie = modelo.Bones["bone_Pie"];
            //trsformaciones
            transMuslo = bone_Muslo.Transform;
            transPantorrilla = bone_Pantorrilla.Transform;
            transPie = bone_Pie.Transform;
            //inicializo la matrz que va a guardar todos los cambios
            boneTransf = new Matrix[modelo.Bones.Count];
        }
```

- Aplicamos método *Update* donde asociaremos un movimiento a una tecla

```
public void update()
        {
            KeyboardState estado = Keyboard.GetState();
            if (estado.IsKeyDown(Keys.Up))
            {
                movimiento1 = movimiento1 + 0.1f;
            }
            if (estado.IsKeyDown(Keys.Down ))
            {
                movimiento2 = movimiento2 - 0.1f;
            }
            if (estado.IsKeyDown(Keys.Left))
            {
                movimiento3 += 0.1f;
            }
```

```
        if (estado.IsKeyDown(Keys.Right))
        {
            movimiento3 -= 0.1f;
        }
    }
```

❖ Sólo falta dibujarlo con método *Draw*

```
public void Draw(Matrix world, Matrix vista, Matrix proyeccion)
    {
        bone_Muslo.Transform = transMuslo * movimiento1;
        bone_Pantorrilla.Transform = transPantorrilla * movimiento2;
        bone_Pie.Transform = transPie * movimiento3;
        modelo.CopyAbsoluteBoneTransformsTo(boneTransf);
        foreach (ModelMesh mesh in modelo.Meshes)
        {
            foreach (BasicEffect efecto in mesh.Effects)
            {
                efecto.EnableDefaultLighting();
                efecto.World = boneTransf[mesh.ParentBone.Index];
                efecto.View = vista;
                efecto.Projection = proyeccion;
            }
            mesh.Draw();
        }
}
```

Dentro del sitio MotorXNA podrás ver el tutorial con una aplicación de rotación con huesos que es donde se encuentra el código original.

http://my.opera.com/motorXna/blog/index.dml/tag/xna%20tutorial

Para continuar con las animaciones que habíamos desarrollado en ***Motion Builder*** te recomendamos entrar al sitio XNA Creators Club Online, ahí encontrarás un ejemplo que le podrás incorporar las animaciones llamadas "takes".
Para eso carga el modelo que ahí te dan y genera más animaciones (take002, 003 etc).

http://creators.xna.com/Headlines/developmentaspx/archive/2007/01/01/Skinned-Model-Sample.aspx

Para incorporarlas agrega código de la siguiente manera:

```
clip2 = skinningData.AnimationClips["Take 002"];
```

y así sucesivamente.....

```
protected override void LoadGraphicsContent(bool loadAllContent)
        {
            if (loadAllContent)
            {
                // Load the model.
                currentModel =
content.Load<Model>("Content/dudeThree");

                // Look up our custom skinning information.
                SkinningData skinningData = currentModel.Tag as
SkinningData;

                if (skinningData == null)
                    throw new InvalidOperationException
                        ("This model does not contain a SkinningData
tag.");

                // Create an animation player, and start decoding an
animation clip.
                animationPlayer = new AnimationPlayer(skinningData);

               clip = skinningData.AnimationClips["Take 001"];
                 clip2 = skinningData.AnimationClips["Take 002"];

                animationPlayer.StartClip(clip);
            }
        }
```

Y añadimos controles dentro de *HandleInput()*

```
private void HandleInput()
        {
            lastKeyboardState = currentKeyboardState;
            lastGamePadState = currentGamePadState;

            currentKeyboardState = Keyboard.GetState();
            currentGamePadState = GamePad.GetState(PlayerIndex.One);

            // Check for exit.
            if (currentKeyboardState.IsKeyDown(Keys.N))
            {
                animationPlayer.StartClip(clip2);
            }
            // Check for exit.
            if (currentKeyboardState.IsKeyDown(Keys.M))
            {
                animationPlayer.StartClip(clip);
            }

            // Check for exit.
            if (currentKeyboardState.IsKeyDown(Keys.Escape) ||
```

```
                AltComboPressed(currentKeyboardState, Keys.F4) ||
                currentGamePadState.Buttons.Back ==
ButtonState.Pressed)
            {
                Exit();
            }
        }
```

Material

- Modelador 3d o software Motion Builder
- XNA Game Studio Express

Desarrollo

Laboratorio

- Genera un sistema de huesos con tu personaje
- En el modelador o Motion Builder haz dos diferentes acciones de animación , como sugerencia:
  - Caminar
  - Correr
- Incorpora las animaciones al código de *Skinning sample*, asignándole una tecla a cada movimiento

Preguntas de Control

1. Describe brevemente la forma en que funcionan los huesos sobre el código de XNA.

Conclusiones

Describe las dificultades encontradas en el desarrollo de la práctica.

Especifica los conocimientos adquiridos en la realización de la misma.

Bibliografía

- http://my.opera.com/motorXna/blog/index.dml/tag/xna%20tutorial
- http://creators.xna.com/Headlines/developmentaspx/archive/2007/01/01/Skinned-Model-Sample.aspx
- http://www.3dyanimacion.com/tutoriales/tutoriales.cfm?estado=ver&titulotutorial=Setup%20de%20personajes%20-%201a%20parte&codigo=5&tutorialID=34
- http://www.tutorialized.com/tutorial/Character-Rigging-in-Maya-Part-1/24521

Práctica

Final

NOMBRE DEL ALUMNO:
INSTRUCTOR:
FECHA:
FASE:
VIDEOJUEGOS CON XNA

Proyecto Final

Objetivos:

- Realizar un Videojuego

Proyecto Final

Como proyecto final creemos que el alumno debe de integrar todas las herramientas vistas en el curso con el fin de evaluar su aprendizaje y poner en práctica sus conocimientos, ingenio y habilidades.

Para ello, se desarrollará una aplicación basada en el Videojuego ***Pizza Commander*** *de Quicksand entertainment (*www.pizzacommander.de), que debe cumplir con los siguientes requerimientos:

Objetivo:

Un personaje que navega sobre un escenario con la tarea de recolectar los ingredientes necesarios para hacer una pizza.

Diseño de Videojuegos

El alumno deberá describir en papel toda la información referente al diseño del Videojuego, esto es, herramientas que utilizará, modeladores, concepción del videojuego, bocetos de personajes, escenarios, objetivos, obstáculos, estrategias y objetos de colisión, texturas, etc.

Pantallas:

El videojuego deberá incluir las siguientes pantallas:

- Menú de Inicio
- Puntuación
- Ayuda
- Créditos

Aquí te damos algunas pantallas de muestra:

Menú Inicio

Puntuación

Interfaz

Interfaz de Usuario:

La interfaz de usuario debe mostrar en el skin o máscara:

- Puntuación
- Cuenta regresiva
- Mapa de navegación
- Objetos recolectados

Ayuda:

La ayuda debe contener toda la información necesaria para entender las reglas, el objetivo y el funcionamiento del juego.

Créditos:

Mostrar créditos de autores.

Interacción:

El personaje deberá recolectar 3 objetos diferentes que serán contabilizados en la interfaz, al terminar los depositará en la base de pasta de pizza.

Colisiones:

Se mostrará al menos un objeto en colisión, esto puede ser, cajas de pizza, obstáculos, etc.

El tiempo de entrega dependerá de cada profesor . Sugerimos 1 mes de desarrollo

Conclusiones

Describe las dificultades encontradas en el desarrollo de la práctica.
¿Es posible incorporar más elementos en tu Videojuego?

Bibliografía

- www.pizzacommander.de

Conclusiones:

El tema de videojuegos, como ya lo habremos visto, es muy amplio y no puede ser agotado en una única investigación, además hasta la realización de este trabajo era muy complicado conseguir la documentación necesaria para su elaboración debido a la reciente aparición de la plataforma XNA como área de conocimiento y de práctica.

Una de los grandes beneficios para los desarrolladores de videojuegos que se localizan en los países en desarrollo, como México, es que con esta herramienta es posible desarrollar juegos de gran calidad a bajo costo, teniendo en cuenta que la mayor parte de la industria de videojuegos está ubicada en países que disponen de alta tecnología, presupuesto e investigación en el área. Sin embargo hay mucha gente interesada, con grandes capacidades en los diferentes campos ya sea animación 3D, programación, diseño, audio, cómputo gráfico, etc, pero hace falta una instancia que los agrupe y que los oriente.

En este sentido, la UNAM juega un papel muy importante, ya sea en la formación del desarrollador, investigador e incluso en el vínculo con la industria, a través de convenios y el suministro de personas altamente capacitadas.

Es también tarea de la Universidad formalizar y brindar programas de educación continua y actualización. En la Facultad de Ciencias, ya se realizó algo en este sentido, a través de la implementación de la materia de videojuegos y la instalación de laboratorios.

Atendiendo a esta necesidad se han diseñado esta serie de prácticas, con el propósito de servir de material de referencia y al mismo tiempo de despertar la inquietud en los nuevos desarrolladores para que amplíen, mejoren y creen nuevos materiales de trabajo para estructurar una posible industria mexicana de videojuegos.

Hay que estar concientes de que una industria fuerte sólo se dará si existen planes concretos a largo plazo. La creación de videojuegos tampoco puede ser trabajo de una sola persona, es por esto que es necesario crear grupos multidisciplinarios (programadores, diseñadores, guionistas, físicos, etc), tomando en cuenta la experiencia de países con una cultura y experiencia exitosa como realizadores de videojuegos.

El mercado actual ofrece muchas oportunidades de desarrollo ya sea en el campo del entretenimiento, cultura, visualización, publicidad, entre otros. El contenido de los videojuegos no solo está orientado al entretenimiento sino que existen desarrolladores que han convertido el videojuegos en una plataforma para concienciar, educar y despertar el interés sobre temas como la ecología, los problemas sociales, humanitarios, entre otros. Estos videojuegos son llamados “de conciencia” y son un nuevo espacio de expresión.

Te invitamos a experimentar, descubrir y sobre todo crear nuevos caminos en el mundo de los videojuegos.

Práctica

ANEXO

Cómo Conectar tu Xbox 360 con XNA Game Studio Express

Cómo conectar tu Xbox 360 con XNA Game Studio Express

Para hacer la conexión deberás seguir los siguientes pasos:

1. **Regístrate a Xbox Live**
   Para poder conectar tu Xbox 360 a la PC es necesario instalar XNA Framework en el disco duro de tu consola a través de Xbox Live Service.
   Este servicio deberá permanecer conectado durante todo el proceso.

   Nota: La conexión entre PC y Xbox 360 no deberá de ser de manera directa, si no buscando que quede sobre la misma red en router o modem, ya que tanto la consola como la PC deben de estar conectadas.

2. **Descarga XNA Game Launcher**

- Navegando en la página de Xbox Live, selecciona **Xbox Live Marketplace** y presiona **A**.
- Selecciona **Games**, presiona **A**.
- Selecciona **All Game Downloads,** presiona **A**.
- Selecciona **XNA Creators Club**.

Para tener la suscripción a "Creators Club" necesitas entrar a Xbox Live Marketplace por $99 por un año o $49 por 4 meses.
Nota: No es necesario que cada alumno tenga una suscripción sino que dentro de la instalación del laboratorio sería indispensable tener una cuenta para que el alumno pueda experimentar.

- Selecciona **Continuar** al mensaje aparecido en pantalla y confirma la descarga, presiona **A**. Obtendrás un mensaje donde confirma que ha sido descargado.
- Selecciona **Memberships,** presiona **A**.
- Dentro de la lista, seleccionar la membresía que deseas comprar y presiona A.

3. **Conexión**

- Dentro de *Games* selecciona **Demos and More,** y presiona **A**.
- Selecciona **XNA Game Launcher** y presiona **A**.

❖ Selecciona **Launch** y presiona **A**. Cuando XNA Game Launcher aparece en la pantalla notarás que hay 3 opciones de menú:

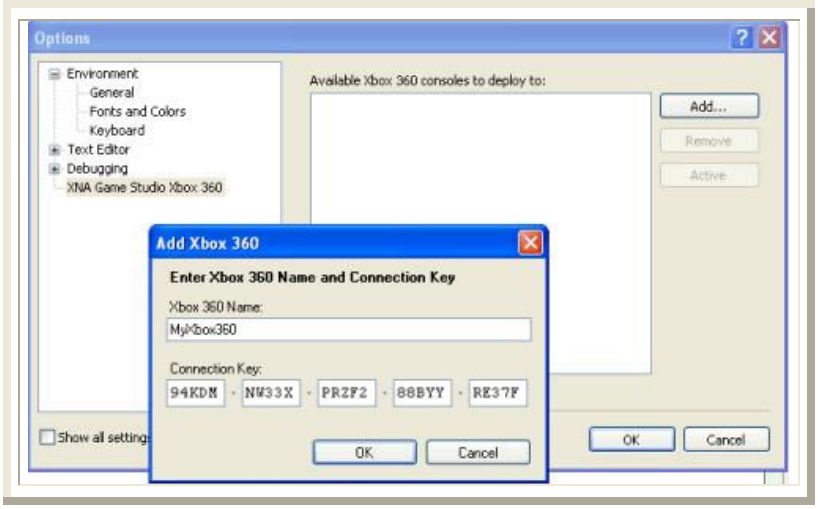
- **My XNA Games**
- **Connect to Computer**
- **Settings**

❖ Seleccionamos **Settings**, presionar **A.**
❖ Seleccionamos **Generate Connection Key** y de nuevo presionamos **A**.
❖ Aparecerá un panel, antes de aceptar deberás introducir este numero en tu XNA Game Studio Express que se encuentra en tu computadora. Esto es, desde *Tools* → *Options* dentro del Visual C# 2005 Express Edition.
❖ De la opción de diálogo selecciona **XNA Game Studio Xbox 360**

❖ Dá un clic sobre **Add**.

❖ Llena los espacios en blanco con el nombre que prefieras y registra la **Connection Key** de tu Xbox y dale clic en **OK**.

❖ En tu Xbox selecciona **Accept New Key** y presiona **A**.



4. Crear un proyecto en Xbox 360

❖ Dentro de la ventana de tu computadora sobre Visual C# 2005 Express Edition selecciona **File →New Project**.
❖ Dentro del cuadro elige **Xbox 360 Game** y escribe el nombre del proyecto.
❖ En el Xbox 360 y Game Launcher selecciona **Connect to Computer** y presiona **A**.
❖ Empezará la conexión.

Está listo para correr tu proyecto.