

Lección 3

En esta lección vamos a adentrarnos en una de las técnicas básicas que dominan la lógica de los videojuegos. Las colisiones.

Los juegos siempre muestran interactividad a través de las interacciones que el personaje del juego tiene con el mundo que le rodea (otros objetos, otros personajes, los límites del mundo, etc.)

Las colisiones pueden manejarse de muchísimas maneras; pero para fines pedagógicos vamos a tratar 2 de las formas más comunes de colisiones en 2D; a saber las colisiones entre rectángulos y las colisiones píxel a píxel.

En esta lección vamos a estudiar la colisión entre rectángulos y la próxima lección estudiaremos la colisión píxel a píxel que es más realista. Para empezar miremos en que consiste una colisión entre rectángulos.

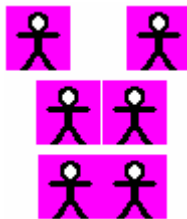


Figura 1

Colisión entre rectángulos

Como podemos ver tenemos 2 imágenes en las cuales nuestro personaje está en la mitad, puede notarse que el personaje es más pequeño que el tamaño total de la imagen. Bien, en un sistema de colisión por rectángulos se detecta la intersección de la caja completa que representa la imagen con otra caja. Es por ello que en la figura 1, la última secuencia representa una colisión, aun cuando los dos personajes no han hecho aun “contacto”, sus cajas sí.

Aunque este tipo de colisiones a simple vista luzca algo rudimentario, aun se pueden hacer juegos bastante entretenidos. Su ventaja radica en su sencillez.

El juego que nos proponemos hacer es un juego de evasión de obstáculos. Este tipo de juegos consisten de un personaje y un conjunto de objetos que están en curso de colisión. El jugador deberá evitar colisionar con los objetos.

Al final de esta lección podremos tener un juego de evasión de obstáculos que luzca así:

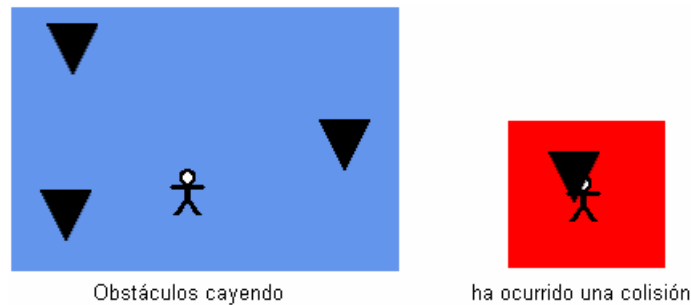


Figura 2
Concepto del juego

Ahora que sabemos en que consiste una colisión, estamos listos para crear nuestro primer juego funcional.

Paso 1: crear el proyecto y cargar las imágenes.

Lo primero que hay que hacer antes de empezar a escribir código es contar con las texturas correspondientes a nuestro personaje y los obstáculos. Estas texturas pueden ser de cualquier tamaño, sin embargo para producir un lindo efecto de transparencia en las áreas vacías de la imagen, se requiere que el color de fondo de esas zonas sea magenta. El magenta proviene de la siguiente combinación RGB (100% rojo, 0% verde, 100% azul). Por ejemplo en Microsoft Paint, podemos producir magenta con la paleta de colores así:



Figura 3
Como combinar el color magenta en Paint
El código RGB¹ del magenta es (255,0,255)

Puedes guardar tus texturas en cualquiera de los siguientes formatos (**.bmp** y **.png**) no como (**.jpg** o **.gif**) pues estos últimos manejan compresión de imagen y por ahora no son compatibles con XNA.

¹ RGB: por sus siglas en ingles es **R**ed, **G**reen and **B**lue (rojo, verde y azul) y representa la combinación de estos 3 colores en valores que van del 0 al 255.

Las texturas que yo utilizare para este tutorial son las de la figura 4, no obstante puedes usar tus propias creaciones.



Figura 4
Un ejemplo de las texturas del juego

A continuación debemos crear un nuevo proyecto en XNA del tipo Windows Game y añadirle las texturas:

Crear un nuevo proyecto windows game.

1. Abre XNA Game Studio Express.
2. Da clic en el menú **File** y luego **New Project** para crear un proyecto.
3. De la lista de plantilla selecciona **Windows Game**.
4. Dale un nombre al juego y escoge una carpeta donde guardarlo.
5. click en **ok**.

Como ya vimos en las lecciones anteriores, XNA ya pone cierta porción de código por nosotros (la plantilla del juego). Para comenzar vamos a cargar nuestras texturas en el proyecto, de esta manera el manejador de contenido de XNA podrá trabajar.

Añadir archivos de textura al proyecto.

1. asegúrate de que puedes ver el explorador de la solución a la derecha de la ventana de XNA. Si no puedes verlo ve al menú **View** y da click en **Solution Explorer**. Cuando aparezca podrás ver la estructura de carpetas de tu proyecto.
2. en el explorador de la solución, da click derecho en el icono del proyecto el cual esta justo abajo del icono de la solución (ver figura 5) y selecciona **add** y luego **new folder**, nombra el nuevo folder como **Content**. Este será el folder raíz de todo el contenido multimedia del juego.
3. da click derecho en el folder **Content** recién creado y selecciona **add** y luego **existing item**. Usando el cuadro de dialogo que aparece navega por tu disco hasta el lugar donde tienes guardadas las 2 texturas. Si las texturas no aparecen, asegúrate de que tiene seleccionado el **File Type** en la posición **Content Pipeline** o **All files**. Selecciona tus 2 texturas y da click en **ok**. Ahora tu proyecto debería verse como en la figura 5.

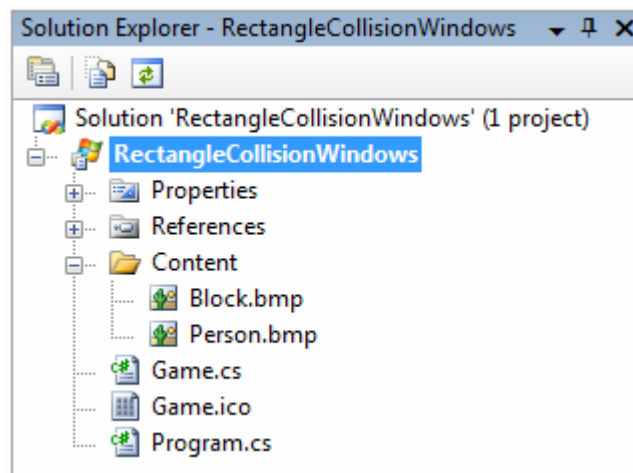


Figura 5
Explorador de soluciones

Ahora si estamos listos para escribir código!

Paso 2: Inicialización y dibujo.

Puede dar una mirada al código que hay en **Game1.cs**, la cual debe de estar en tu pantalla al cargar el proyecto. Lo primero que queremos hacer es escribir el código necesario para dibujar las texturas; no obstante, primero debemos cargar las texturas y establecer un par de variables que nos ayudaran a posicionar nuestro personaje.

1. doble clic en el archivo game1.cs en el explorador de soluciones.
2. añade los siguientes atributos dentro de la clase **Game1**.

```
// Las imagenes a dibujar
Texture2D texturaPersona;
Texture2D texturaBala;

// Las imagenes son dibujadas con el objeto spriteBatch
SpriteBatch spriteBatch;

// Un vector para las pocisiones X,Y de la persona
Vector2 posicionPersona;

// Una lista de vectores para las pocciones X,Y de las Balas
List<Vector2> posicionesBalas = new List<Vector2>();
```

Los vectores Vector2 son usados en las graficas 2D por que contienen 2 coordenadas (X,Y) que ayudan a posicionar los objetos en el plano cartesiano. Una lista, como su nombre lo indica, no es más que una colección de otros objetos; en este caso es una colección de Vectores

3. debemos inicializar estas variables, para ello contamos con los métodos **Initalize** y **LoadGraphicsContent**. Las texturas se inicializaran en el

segundo, y el resto de la lógica de inicialización la hacemos en el primero. Pon dentro de esos 2 métodos el siguiente código (cambios en negrita).

```
protected override void Initialize()
{
    base.Initialize();

    // Iniciar el personaje en el centro de abajo de la pantalla
    // puedes probar con diferentes valores hasta dar con el que
    // mejor se te ajuste.
    posicionPersona.X = 100;
    posicionPersona.Y = 500;
}

protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent) {
        // Cargar las texturas
        texturaBala = content.Load<Texture2D>("Content/Block");
        texturaPersona = content.Load<Texture2D>("Content/Person");

        // Crear el sprite batch para dibujar las texturas
        spriteBatch = new SpriteBatch(graphics.GraphicsDevice);
    }
}
```

4. Ya tenemos las texturas cargadas, ahora es tiempo de escribir el código necesario para dibujarlas en la pantalla; para esto modificaremos el método Draw Para que luzca así:

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

    spriteBatch.Begin();

    // Dibujar persona
    spriteBatch.Draw(texturaPersona, posicionPersona, Color.White);

    // Dibujar balas
    foreach (Vector2 posicionBala in posicionesBalas) {
        spriteBatch.Draw(texturaBala, posicionBala, Color.White);
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```

La instrucción **foreach** es una estructura de código repetitiva que tiene el propósito de recorrer colecciones. Tiene una estructura de la siguiente manera:

Para cada (elemento del tipo X in la colección Y) haga.
{// Instrucciones a repetir}

En nuestro caso el **foreach** va a recorrer la colección *posicionesBalas* usando los vectores que hay en su interior para pintar las balas una a una.

Hay que destacar que si la colección tiene 100 elementos este código se repetirá 100 veces y si no tiene elementos, no se ejecutara ni una sola vez.

5. si compilamos y ejecutamos el proyecto en este punto (Botón de play o F5) deberías estar viendo al personaje, pero ¿y donde están las balas?

Paso 3: Hacer que todo se mueva.

No hay balas en el juego cuando lo ejecutas, por que la lista de balas esta vacía, necesitamos crear toda la lógica necesaria para crear y animar las balas. Y ya que estamos animando, ¿Por qué no animar el personaje de una vez?

1. Por simplicidad de este tutorial, vamos a hacer que tanto el personaje como las balas se muevan a velocidades constantes. El número y las posiciones de las balas van a ser determinadas de manera aleatoria. Para lograr esto agrega las siguientes líneas en el inicio de la clase **Game1** (las líneas nuevas están en negrilla).

```
// Un vector para las posiciones X,Y de la persona
Vector2 posicionPersona;
int velocidadPersona = 5;

// Una lista de vectores para las pocisiones X,Y de las Balas
List<Vector2> posicionesBalas = new List<Vector2>();
double provabilidadBalas = 0.05; // es decir 5%
int velocidadBalas = 2;

Random aleatorio = new Random();
```

Estos valores los puse yo, pero tú puedes modificarlos a tu gusto. Obviamente a mayor velocidad de las balas y mayor probabilidad de que aparezcan más balas juntas, mayor dificultad le darás a tu juego.

2. ahora modifica el método **update** para que luzca así:

```
protected override void Update(GameTime gameTime)
{
    // Objeto que nos permite escuchar el teclado
    KeyboardState keyboard = Keyboard.GetState();

    // Dar la posibilidad de salir del juego
    if (keyboard.IsKeyDown(Keys.Escape)) {
        // si se preciona la tecla escape se termina el juego
        this.Exit();
    }

    // Mover el personaje con las flechas derecha e izquierda
    if (keyboard.IsKeyDown(Keys.Left)) {
        posicionPersona.X -= velocidadPersona;
    }
}
```

```
if (keyboard.IsKeyDown(Keys.Right)) {  
    posicionPersona.X += velocidadPersona;  
}  
  
// aparecen nuevas balas según la probabilidad  
if (aleatorio.NextDouble() < probabilidadBalas) {  
    float x = (float)aleatorio.NextDouble() *  
        Window.ClientBounds.Width;  
    posicionesBalas.Add(new Vector2(x, 0));  
}  
  
// actualizar cada bala  
for (int i = 0; i < posicionesBalas.Count; i++) {  
    // animar la bala cayendo  
    posicionesBalas[i] = new Vector2(posicionesBalas[i].X,  
        posicionesBalas[i].Y + velocidadBalas);  
}  
  
base.Update(gameTime);  
}
```

En el bloque de código anterior está la estructura repetitiva **for** la cual tiene un comportamiento parecido al **foreach** que vimos anteriormente. En realidad el **foreach** es un caso especial de la estructura **for**.

La estructura **for** es una estructura repetitiva que se puede usar de manera genérica para repetir un conjunto de instrucciones *n* veces, mientras se cumpla una condición. La sintaxis es así:

Para (un valor inicial; una condición; un incremento sobre el valor) haga
 { // Instrucciones a repetir }

En nuestro caso el **for** va a tener un contador que inicia en cero y va a repetirse mientras el contador sea menor que el número de elementos en la colección de balas, con un incremento de uno en uno por vez.

3. Estoy seguro de que hay bastante código aquí que te resulta confuso, es por eso que vamos a repasarlo con cuidado para ver que hace cada parte.
 - a. Primero obtener el manejador del teclado.
 - b. Dar la posibilidad de terminar la aplicación.
 - c. Mover el personaje con las flechas del teclado.
 - d. Crear nuevas balas en posiciones aleatorias, la probabilidad define que tan a menudo aparecen, y la velocidad que tan rápido caen.
 - e. Animar cada bala cayendo.

Puedes compilar el juego y ejecutarlo en este punto. Verás al personaje al que ahora puedes mover, y las balas cayendo.

Paso 4: Manejar las colisiones con la pantalla.

Si ya has jugado un poco habrás notado al menos 2 cosas: primero, no pasa nada cuando las balas tocan al personaje, y segundo el personaje se puede salir de la pantalla.

Hay un tercer aspecto no tan obvio que también debemos corregir y es que las balas que caen nunca son borradas (siguen cayendo indefinidamente). Si dejásemos el juego corriendo por un periodo prolongado de tiempo, eventualmente el número de balas sería tan grande que la memoria del computador se agotaría! Para remediar esto, simplemente borraremos las balas al salir de la pantalla.

1. modifica el código del método **update** para que luzca de la siguiente manera (las líneas nuevas están en negrilla):

```
protected override void Update(GameTime gameTime)
{
    // Objeto que nos permite escuchar el teclado
    KeyboardState keyboard = Keyboard.GetState();

    // Dar la posibilidad de salir del juego
    if (keyboard.IsKeyDown(Keys.Escape)) {
        // si se presiona la tecla escape se termina el juego
        this.Exit();
    }

    // Mover el personaje con las flechas derecha e izquierda
    if (keyboard.IsKeyDown(Keys.Left)) {
        posicionPersona.X -= velocidadPersona;
    }
    if (keyboard.IsKeyDown(Keys.Right)) {
        posicionPersona.X += velocidadPersona;
    }

    // Prevenir que el personaje se salga de la pantalla
    // el método clamp ajusta el rango de valores que
    // posicionPerona.X puede tomar
    posicionPersona.X = MathHelper.Clamp(posicionPersona.X,
        0, Window.ClientBounds.Width - texturaPersona.Width);

    // aparecen nuevas balas según la probabilidad
    if (aleatorio.NextDouble() < probabilidadBalas) {
        float x = (float)aleatorio.NextDouble() *
            Window.ClientBounds.Width;
        posicionesBalas.Add(new Vector2(x, 0));
    }

    // actualizar cada bala
    for (int i = 0; i < posicionesBalas.Count; i++) {
        // animar la bala cayendo
        posicionesBalas[i] = new Vector2(posicionesBalas[i].X,
            posicionesBalas[i].Y + velocidadBalas);

        // eliminar las balas cuando salen de la pantalla
        if (posicionesBalas[i].Y > Window.ClientBounds.Height) {
            posicionesBalas.RemoveAt(i);
        }
    }
}
```



```
        // decrecemos i, por que hay una bala menos
        i--;
    }
}

base.Update(gameTime);
}
```

Este par de nuevas líneas de código previenen que nuestro personaje se salga de la pantalla (para que nos hagan trampa) y que la memoria del computador se llene innecesariamente.

A pesar de que no lo parezcan estas son las 2 primeras formas de colisión que has manejado (si, el personaje contra los bordes laterales de la pantalla y de las balas contra el borde inferior de la misma).

2. compila y ejecuta el proyecto. Ahora no puedes sacar el personaje de la pantalla, y las balas no pondrán lento el computador eventualmente.

Paso 5: Manejar las colisiones entre rectángulos.

Ahora nos podemos preparar para la parte que inicio este tutorial, la colisión entre rectángulos, la cual es la última pieza del rompecabezas que nos falta para terminar nuestro primer juego de colisiones.

Por el alcance de este tutorial, no vamos a realizar acciones complejas cuando ocurra la colisión², sino que simplemente pondremos el color de fondo en rojo.

1. añade la siguiente línea al inicio de la clase **Game1**.

```
// Variable que indica si hay o no colisión
bool colision = false;
```

Una variable del tipo bool (booleana) es un tipo básico de C# que solo puede tomar 2 posibles valores: Verdadero (**true**) o Falso (**false**).

2. modifica el método **Draw**. Para que luzca así (las nuevas líneas están en negrilla):

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice device = graphics.GraphicsDevice;

    // cambiar el fondo a rojo si hay colisión
    if (colision){
        device.Clear(Color.Red);
    }else{
        device.Clear(Color.CornflowerBlue);
    }
}
```

² Algunas acciones que haremos en próximos tutoriales, serian las de producir un sonido y cambiar la imagen del personaje cuando halla la colisión.

```
spriteBatch.Begin();

// Dibujar persona
spriteBatch.Draw(texturaPersona, posicionPersona, Color.White);

// Dibujar balas
foreach (Vector2 posicionBala in posicionesBalas){
    spriteBatch.Draw(texturaBala, posicionBala, Color.White);
}

spriteBatch.End();

base.Draw(gameTime);
}
```

3. Ahora hay que determinar cuando una de las balas que cae colisiona con el personaje. La colisión de rectángulos que es la que utilizaremos en este tutorial, es manejada por el marco de trabajo de XNA gracias al método **Rectangle.Intersects** de la clase Rectangle. Modifica el método update para que luzca así (las nuevas líneas están en negrilla):

```
protected override void Update(GameTime gameTime)
{
    // Objeto que nos permite escuchar el teclado
    KeyboardState keyboard = Keyboard.GetState();

    // Dar la posibilidad de salir del juego
    if (keyboard.IsKeyDown(Keys.Escape)) {
        // si se presiona la tecla escape se termina el juego
        this.Exit();
    }

    // Mover el personaje con las flechas derecha e izquierda
    if (keyboard.IsKeyDown(Keys.Left)) {
        posicionPersona.X -= velocidadPersona;
    }
    if (keyboard.IsKeyDown(Keys.Right)) {
        posicionPersona.X += velocidadPersona;
    }

    // Prevenir que el personaje se salga de la pantalla
    // el método clamp ajusta el rango de valores que
    // posicionPerona.X puede tomar
    posicionPersona.X = MathHelper.Clamp(posicionPersona.X,
        0, Window.ClientBounds.Width - texturaPersona.Width);

    // aparecer nuevas balas según la probabilidad
    if (aleatorio.NextDouble() < probabilidadBalas){
        float x = (float) aleatorio.NextDouble() *
            Window.ClientBounds.Width;
        posicionesBalas.Add(new Vector2(x, 0));
    }
}
```

```
// obtener el rectángulo de la persona
Rectangle rectanguloPersona =
new Rectangle((int) posicionPersona.X, (int) posicionPersona.Y,
    texturaPersona.Width, texturaPersona.Height);

// actualizar cada bala
colision = false;
for (int i = 0; i < posicionesBalas.Count; i++){
    // animar la bala cayendo
    posicionesBalas [i] = new Vector2(posicionesBalas [i].X,
        posicionesBalas[i].Y + velocidadBalas);

    // obtener el rectángulo de la bala
    Rectangle rectanguloBala =
        new Rectangle((int) posicionesBalas[i].X,
            (int) posicionesBalas[i].Y,
            texturaBala.Width, texturaBala.Height);

    // evaluar colisión con el personaje
    if (rectanguloPersona.Intersects(rectanguloBala)){
        colision = true;
    }

    // eliminar las balas cuando salen de la pantalla
    if (posicionesBalas [i].Y > Window.ClientBounds.Height){
        posicionesBalas.RemoveAt(i);
        // decrecemos i, por que hay un bloque menos
        i--;
    }
}

base.Update(gameTime);
}
```

El código que acabamos de escribir, evalúa constantemente si el rectángulo que representa la figura del personaje se intercepta con el rectángulo de alguna de las balas que caen.

4. bueno eso es todo! Ya tienes el juego terminado, puedes compilarlo y empezar a jugar.

Felicitaciones! ya comprendes uno de los aspectos más básicos de la programación de videojuegos gracias a este pequeño tutorial. Con base en esto no hay límites a lo que puedes hacer.

Espera en nuestro próximo tutorial: colisiones 2D píxel a píxel.

Algunas ideas para mejorar el juego:

- Añade un contador de las veces que el personaje es golpeado.
- Modifica la velocidad de las balas para hacer más difícil el juego.
- Has que a medida que pasa el tiempo, poco a poco las balas caigan mas rápido.

- Añade un poder especial que caiga de tanto en tanto y que al colisionar con el personaje le de 5 segundos de invisibilidad.

Para los curiosos:

Una lección extra de programación que podemos sacar de este tutorial son las estructuras repetitivas. En términos generales se denominan estructuras repetitivas a las estructuras que permiten contener dentro de ellas un conjunto de instrucciones que deben hacerse repetidamente mientras una condición sea verdadera; es decir que las instrucciones pueden ejecutarse 0 o n veces.

Los tipos de estructuras repetitivas para C# son:

Sintaxis	Explicación
<pre>while(condición){ // instrucciones }</pre>	<p>Un conjunto de instrucciones se ejecutaran mientras la condición inicial sea verdadera; puede que no se ejecute nunca, como puede que se ejecute de manera infinita si la condición no se torna falsa nunca. Es por eso que dentro de las instrucciones repetitivas se debe incluir una forma de invalidar la condición en algún momento si no se quiere quedar en el while de manera indefinida ejemplo:</p> <pre>int valor = 1; while (valor < 10){ // instrucciones valor++; } // Ejecutara las instrucciones 9 veces.</pre>
<pre>do{ // instrucciones }while(condición)</pre>	<p>Igual que el while; la única diferencia es que la condición se evalúa el final de cada ciclo y no al inicio. Esto significa que las instrucciones se ejecutaran al menos una vez (la primera) y de allí en adelante mientras la condición sea verdadera.</p> <pre>Bool condicion = false; do{ // instrucciones }while(condicion) /* ejecutará las instrucciones una vez así la condición sea falsa por que se evalúa al final; sin embargo no lo hará mas veces*/</pre>
<pre>for(valor inicial; condición; incremento){ // instrucciones }</pre>	<p>Un conjunto de instrucciones que se repetirán tantas veces como se cumpla la condición.</p> <pre>for(int valor=0; valor<10; valor++){ // instrucciones } // repetirá las instrucciones 10 veces</pre>
<pre>foreach(elemento del tipo X in colección Y){ // instrucciones }</pre>	<p>Se usa para recorrer colecciones iterando sobre cada uno de los elementos que esta</p>

}	<p>contienen.</p> <pre>ArrayList lista = new ArrayList(); /* introduzca elementos por ejemplo de tipo cadena en la lista*/ lista.add("Hola "); lista.add("esta es una "); lista.add("cadena larga"); string cadenaTotal = ""; // recorrer la colección foreach(string cadena in lista){ cadenaTotal += cadena; } /* al terminar el foreach, cadenaTotal tendría el siguiente contenido: "Hola esta es una cadena larga" */</pre>
---	---