Tema 1. Contacto con la creación de videojuegos

1.1. Bibliotecas para juegos y motores de juegos

Existen multitud de bibliotecas y de motores que se pueden utilizar para crear juegos. Por lo general, se tiende a usar la palabra "**biblioteca**" cuando sólo proporciona soporte para tareas muy básicas, como mostrar imágenes, reproducir sonidos o comprobar el estado del teclado y otros dispositivos de entrada. Por el contrario, se suele usar la palabra "**motor**" cuando añade otras funcionalidades de nivel más alto, como soporte para "físicas" (gravedad, colisiones, sistemas de partículas, etc), animaciones, plantillas de juegos o de elementos (personajes con lógicas asociadas, menús, etc). Aun así, esta separación no siempre es clara, porque muchas de las bibliotecas básicas se han ido enriqueciendo con el tiempo para añadir más funcionalidades, o bien existen módulos de terceros que permiten ampliarlas y acelerar el desarrollo de programas más complejos.

Algunas de las bibliotecas para juegos más habituales son:

- **SDL** es una biblioteca muy empleada para juegos en 2D, emuladores y software que necesite un acceso directo al hardware. Su primera versión es de 1998. Está creada en C, por lo que es directamente utilizable desde C y C++, pero existen "bindings" para otros lenguajes como C# y Python. **Pygame** es una biblioteca para Python que se apoya en SDL.
- **Allegro** cubre el mismo segmento que SDL, es más antigua (de principios de los 90) pero se sigue manteniendo. Eso sí, su cuota de mercado es menor. También está creada en C y tiene "bindings" para otros lenguajes.
- **XNA**, de Microsoft, era un conjunto de herramientas para facilitar la creación de juegos para Windows y XBox usando C#. Cuando Microsoft abandonó su desarrollo, surgieron alternativas open source para seguir empleando ese modelo, como **MonoGame** y FNA. MonoGame será la primera herramienta que usemos.
- **OpenGL** es una biblioteca gráfica de muy bajo nivel, orientada principalmente a 3D (aunque también se puede usar en 2D), y relativamente poco amigable.
- **LibGDX** es una biblioteca creada (principalmente) en Java, inicialmente para desarrollos en Android, pero que también permite crear juegos de escritorio. Comenzó en 2014.
- **Cocos2d** es una biblioteca de código abierto, creada inicialmente para Python pero de la que luego se han hecho muchos "forks", para ObjectiveC (iPhone), C++, Ruby, Java, C#, JavaScript...
- **Kivy** es para Python, de código abierto, pensada principalmente para dispositivos móviles, aunque también permite desarrollar software de escritorio.

Y como ejemplos de motores y herramientas:

- **Unity3D** es un motor de juegos multiplataforma 3D y 2D, propietario, creado en C++ pero que usa C# como lenguaje de script. Su origen es del año 2005 y se permite su uso libre para usuarios/empresas que generen menos de 100.000 dólares al año, debiéndose pagar a partir de ese punto. Será la herramienta en la que nos centraremos.
- **Unreal Engine** es un motor de juegos creado inicialmente por Epic Games para el juego FPS Unreal (1998) y evolucionado posteriormente. Está creado en C++ y orientado sobre todo a la creación de juegos 3D, aunque también se puede emplear para 2D y otras temáticas de juego distinto de los "shooters" en primera persona. Permite generar ejecutables para multitud de plataformas. Su código libre es descargable y su modelo de negocio se basa en royalties generados por los programas que se creen usándolo.
- **idTech** es el nombre genérico que se da a una serie de motores creados por idSoftware para sus juegos Doom, Quake y secuelas posteriores. Las primeras versiones de estos motores son de código abierto (y se pueden descargar desde la cuenta de GitHub de idSoftware), pero las más recientes no.
- **Panda3D** es un motor de código abierto, creado por Disney y la Carnegie Mellon University, desarrollado en C++ pero que usa Python como lenguaje de script. En general, resulta mucho menos amigable que otros como Unity.
- **Construct** permite crear juegos en HTML5 (lo que supone que también se pueden convertir a "aplicaciones híbridas" para hacerlos funcionar en Android e iOS, o incluso en escritorio), y usa una aproximación "visual", que permite indicar los eventos a los que debe responder el programa, en vez de teclear órdenes. La primera versión era de código abierto, pero la actual es comercial.
- **CoronaSDK** es otro producto propietario, de código cerrado, pero que se puede usar "gratis" en la mayoría de sus funcionalidades, si bien algunas avanzadas son de pago. Usa LUA como lenguaje de script y permite generar aplicaciones de escritorio y móviles.
- **GameMaker Studio** también es un producto propietario, que usa su propio lenguaje de script (bastante parecido a LUA o BASIC) y que permite exportar para múltiples plataformas. La versión gratuita sólo funciona durante 30 días, y a partir de ese momento se debe pagar según las plataformas de destino (por ejemplo, actualmente son $39 al año o $99 de por vida para Windows y Mac, $149 de por vida para HTML5, $149 de por vida para Android...).

Pero hay muchísimos más, tanto de código abierto como comerciales... Puedes ver una lista mucho más exhaustiva en:

https://en.wikipedia.org/wiki/List_of_game_engines

y otra un poco más compacta en:

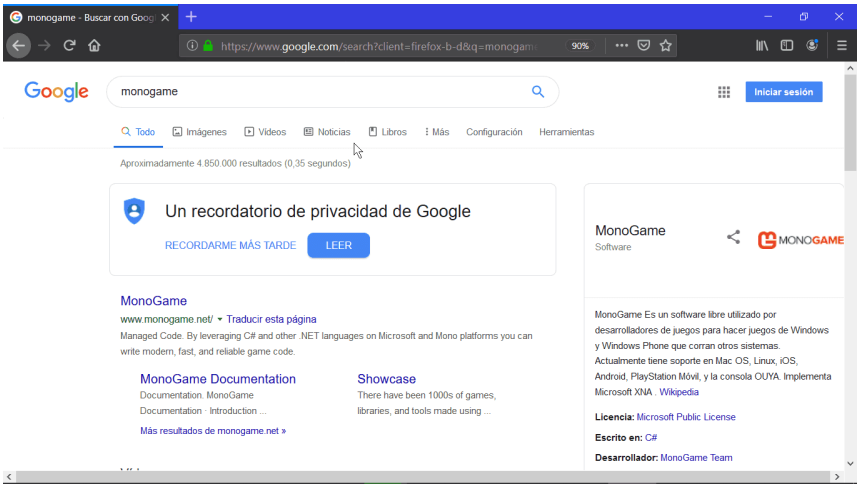
https://en.wikipedia.org/wiki/LibGDX#External_links

1.2. ¿Qué es MonoGame?

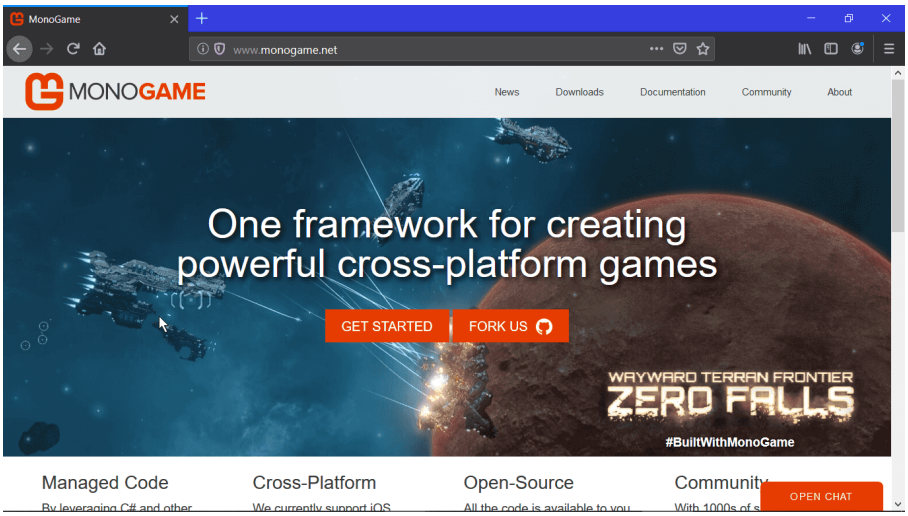
En este primer tema vamos a tener un contacto con una biblioteca para creación de juegos usando el lenguaje C#. Se trata de MonoGame. Es de un nivel algo más alto que otras como SDL, porque incluye un cierto control del bucle de juego, pero aun así deja mucho trabajo en manos del programador. Es un proyecto de código abierto, que nació a partir del proyecto XNA, que creó Microsoft en 2004 como ayuda para desarrolladores indie, tanto para Windows como para Xbox. XNA dejó oficialmente de mantenerse en 2013, y en ese momento surgieron alternativas open source basadas en su modo de trabajo, como MonoGame y FNA. Mientras FNA busca compatibilidad total con XNA, pero obliga a tener una instalación original de XNA y usar sus herramientas originales, MonoGame no busca esa compatibilidad total y sí incluye ciertas herramientas auxiliares.

1.3. Descarga e instalación de MonoGame

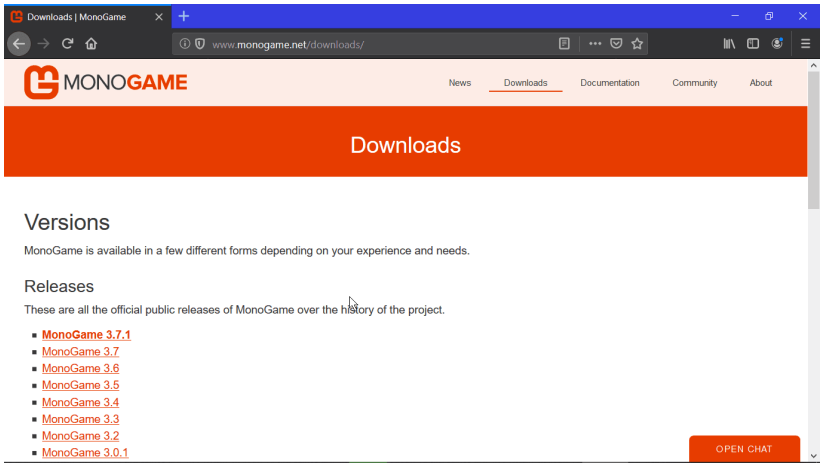
Basta teclear "monogame" en nuestro buscador favorito:



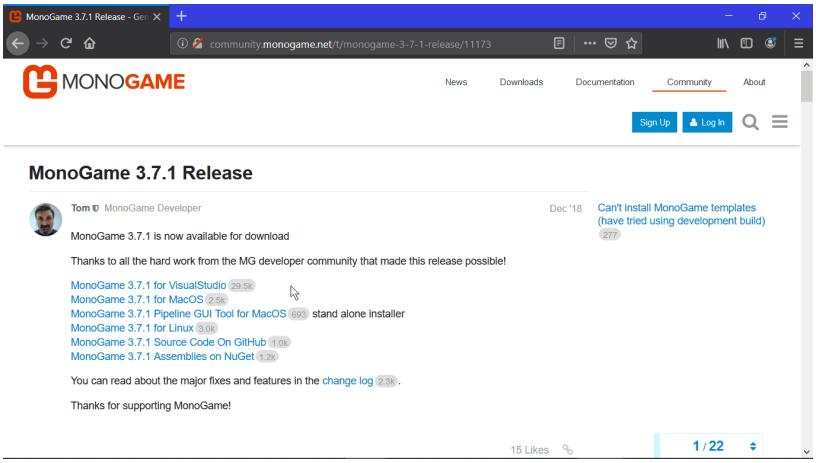
Y llegaremos a su página oficial, monogame.net:



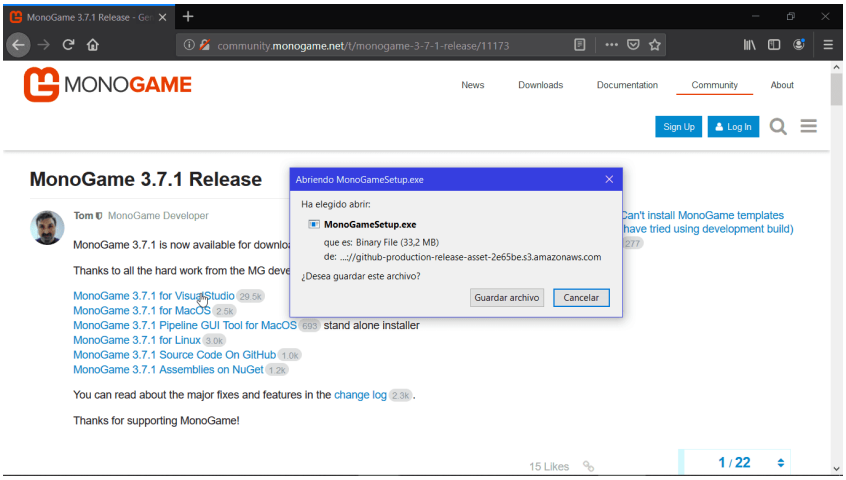
En ella hay un apartado de Descargas (downloads):



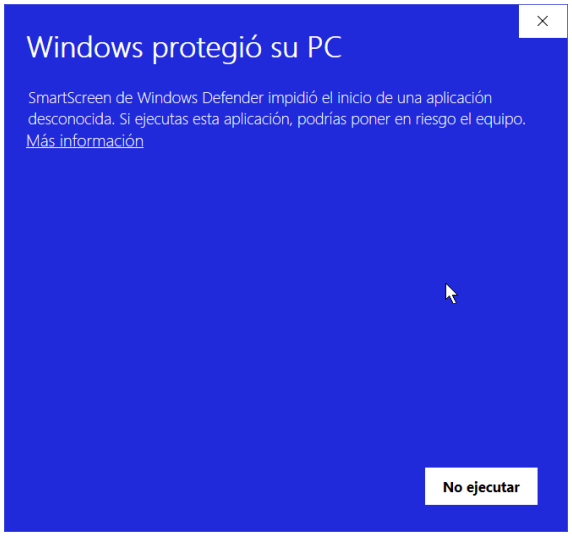
En el momento de escribir este texto, la última versión disponible es la 3.7.1, que se puede descargar para Visual Studio (lo que supondrá la alternativa más sencilla para Windows) y para MacOS:



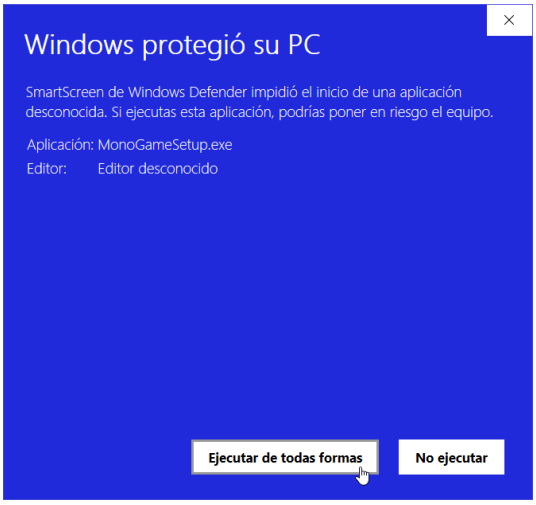
La descarga para Visual Studio es un fichero de apenas 35 MB:



Quizá aparezca un aviso de Windows avisando de que es un fichero desconocido y, por tanto, peligroso:

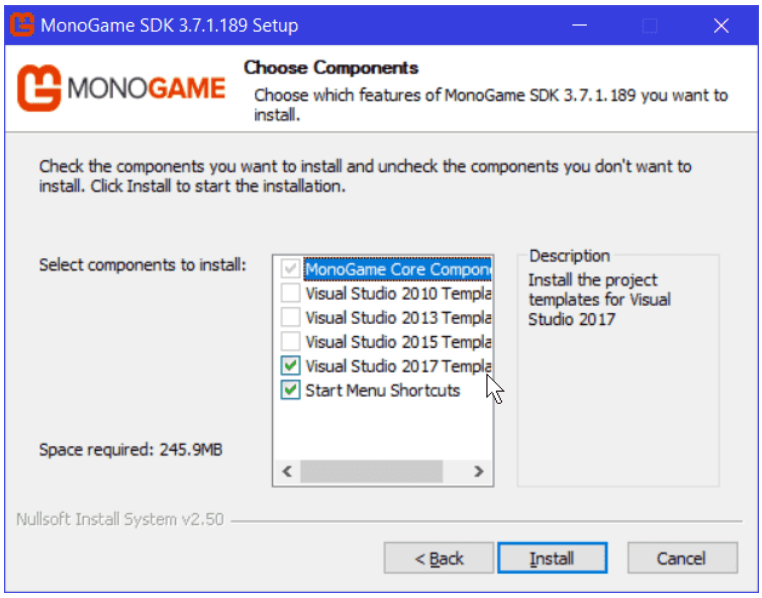


Pero si lo hemos descargado desde su página oficial, podremos seguir adelante:

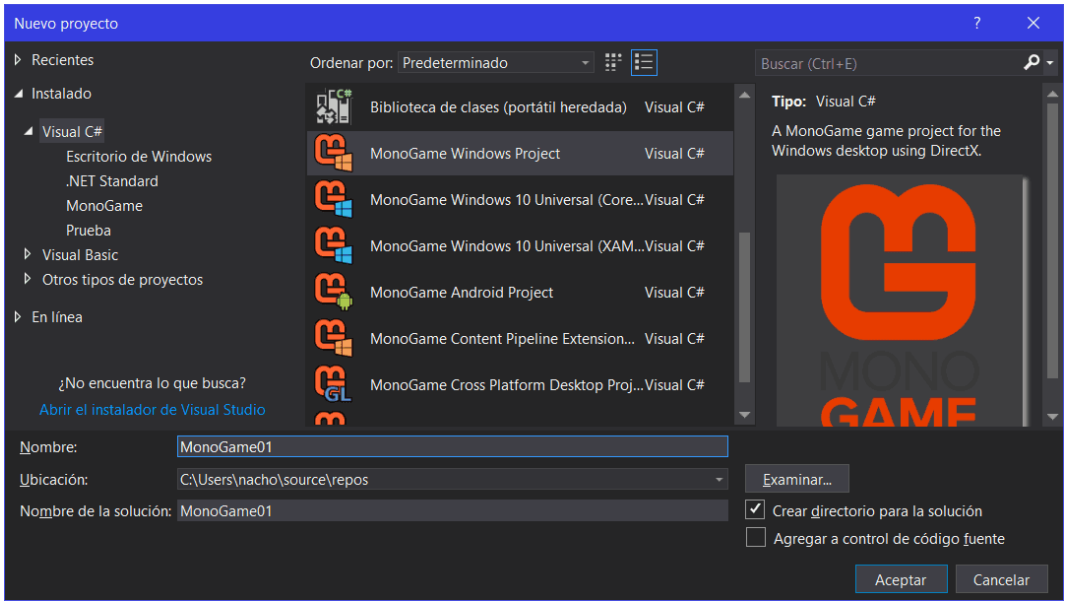


Y el asistente de instalación debería consistir poco más que en pulsar el botón Siguiente un par de veces:

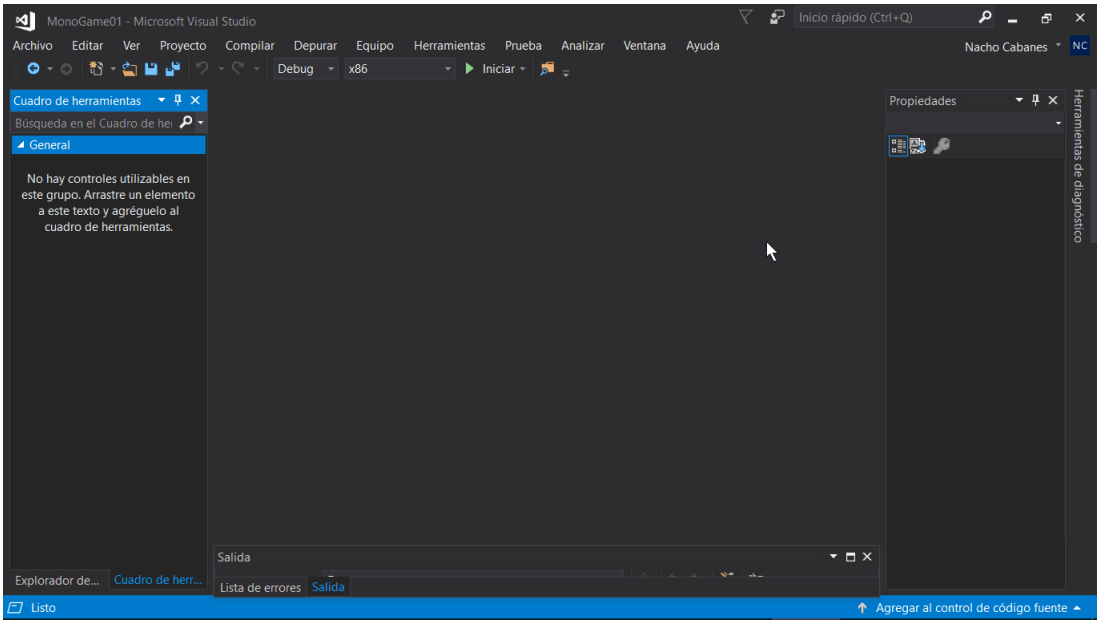
El propio asistente debería detectar qué instalaciones de Visual Studio tenemos y proponernos instalar las plantillas para cada una de ellas, desde VS2010 hasta (actualmente) VS2017. Eso supone un primer escollo: Visual Studio 2017 ya no se puede descargar desde la página de Microsoft, y MonoGame no ofrece (aparentemente) soporte para Visual Studio 2019. Un poco más adelante veremos cómo solucionarlo "de forma artesanal" (hasta que, se supone que dentro de poco, MonoGame soporte oficialmente Visual Studio 2019). En este primer instante vamos a suponer que estamos empleando la versión 2017 o alguna anterior:



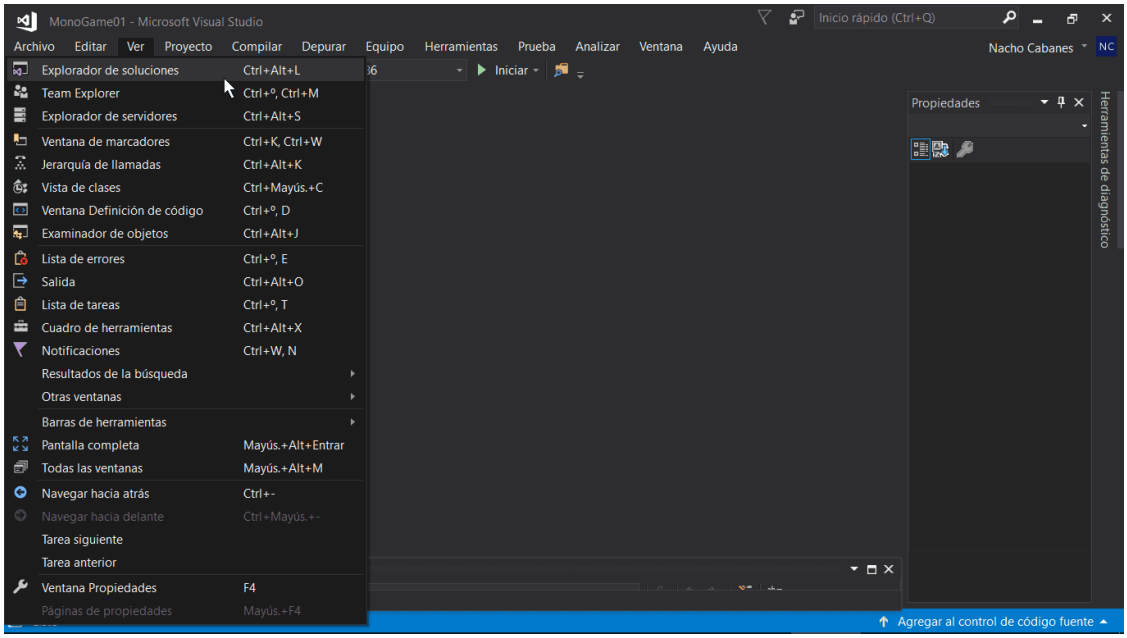
Tras completar la instalación, si entramos a Visual Studio, nos aparecerán nuevas plantillas, relacionadas con MonoGame. Podemos elegir la llamada "Monogame Windows Project":



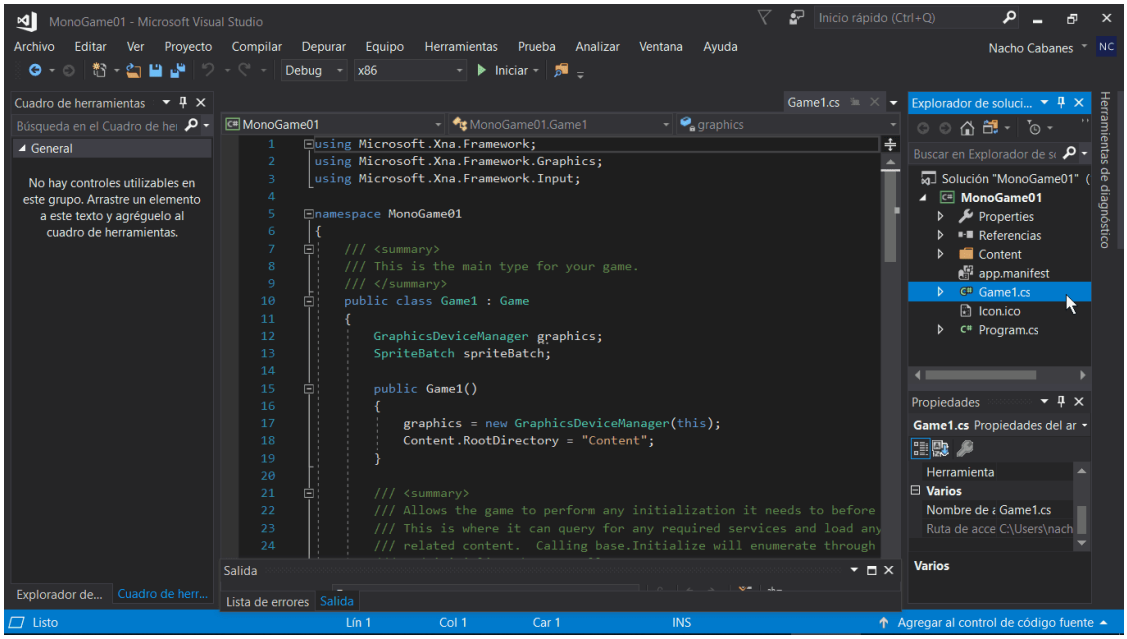
Podría ocurrir que viéramos la pantalla de edición vacía, sin ningún código fuente en primer plano:



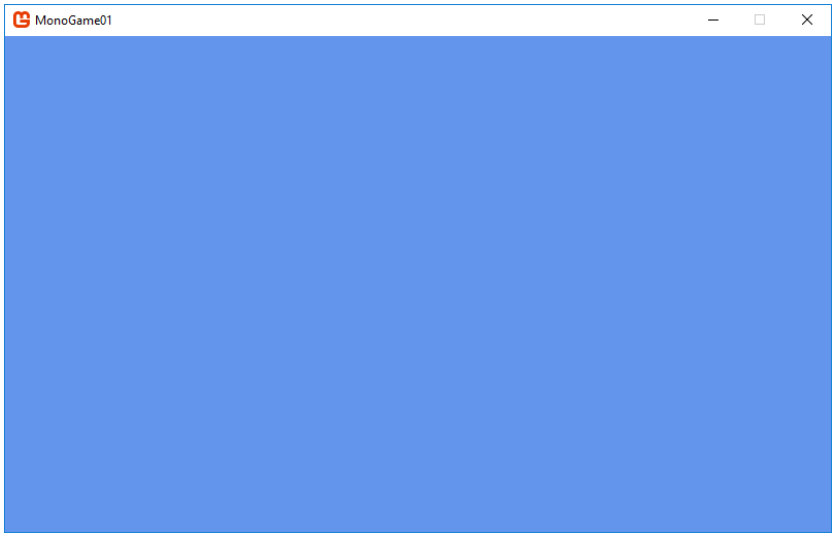
Y en ese caso deberíamos mirar si está abierto el "Explorador de Soluciones" en la parte derecha de la pantalla, o abrirlo desde el menú "Ver" en caso contrario:



Entonces ya aparecerán un par de ficheros: "Program.cs", que actúa simplemente como lanzador, y "Game.cs", que contiene un esqueleto de juego que iremos completando:

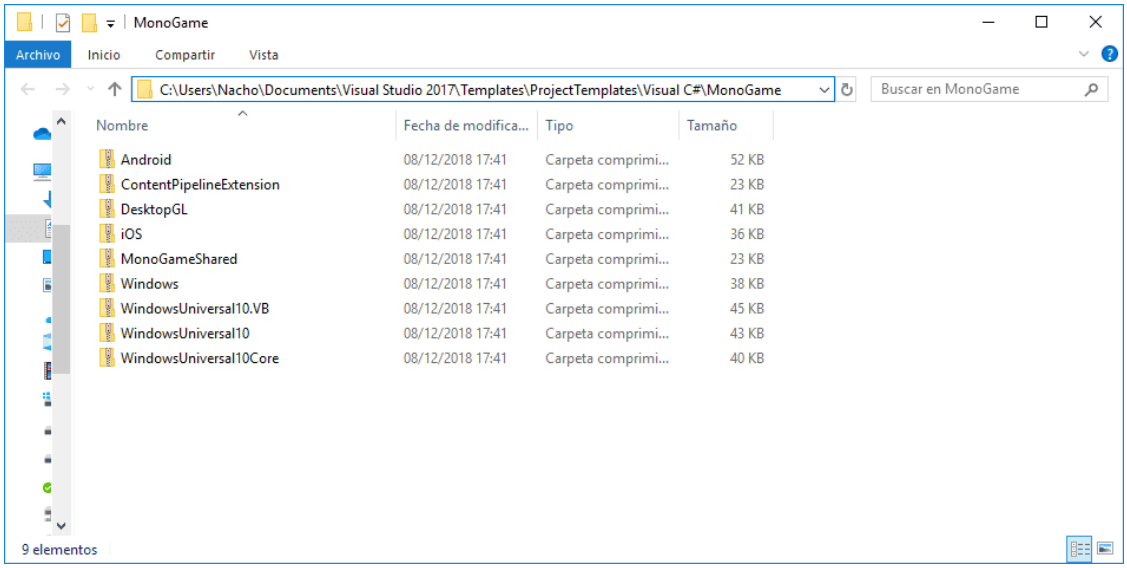


Y si pulsamos el botón Iniciar, veremos que aparece una ventana con fondo azul:



Si estás en **Visual Studio 2019** y todavía no hubiera plantillas de MonoGame para él, puedes optar por hacer una instalación de MonoGame por defecto (que no te creará plantillas para VS2019) y luego "imitar" lo que hace la instalación para VS2017, que es crear las plantillas en la carpeta

```
C:\Users\[Tu usuario]\Documents\Visual Studio 2017\Templates\ProjectTemplates\
Visual C#\MonoGame
```



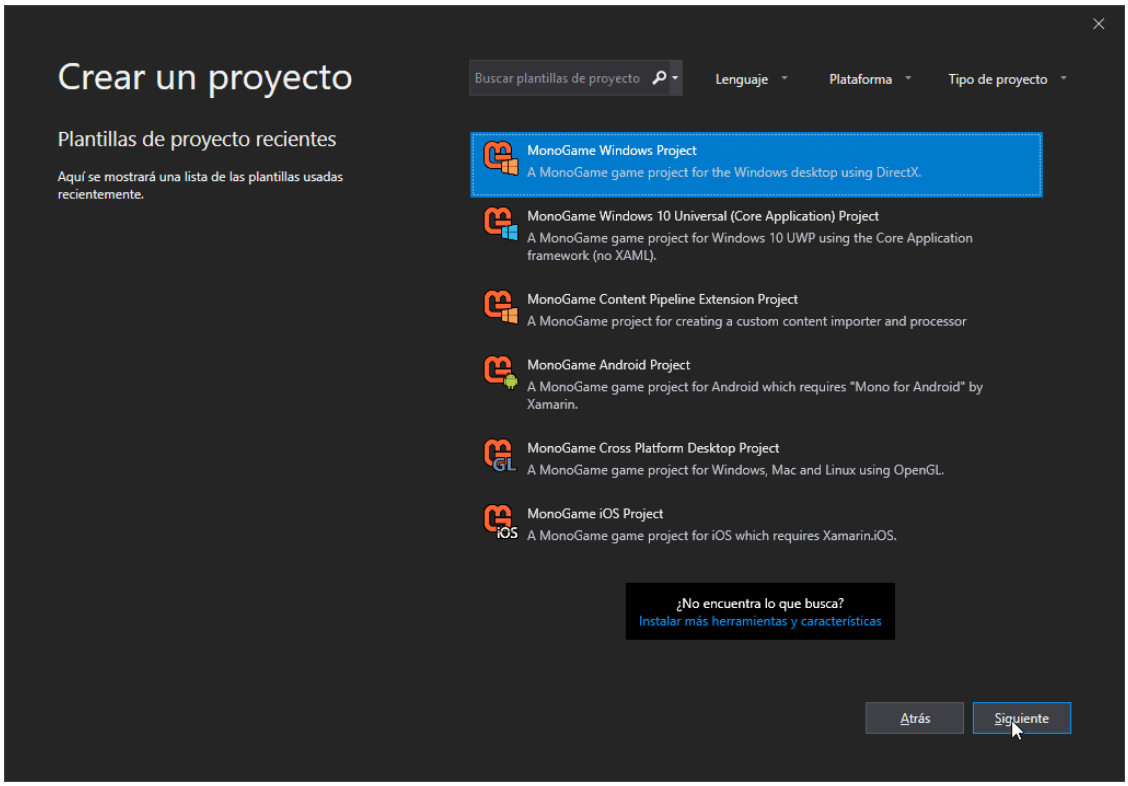
Por tanto, deberás, crear una carpeta llamada:

```
C:\Users\[Tu usuario]\Documents\Visual Studio 2019\Templates\ProjectTemplates\
Visual C#\
```

Y en ella descomprimir, respetando subdirectorios, el contenido del fichero:

https://iessanvicente.com/recursos/MonoGame.zip

Si ahora vuelves a entrar a Visual Studio 2019 y pides crear un proyecto nuevo, ya deberían aparecerte las plantillas correspondientes a MonoGame:



Nota: En el momento de escribir y revisar este texto (sept.2019), la carencia de esas plantillas es una queja frecuente entre los usuarios de MonoGame, desde hace más de 5 meses (abr.2019). Tienes más detalles y una descarga adicional en:

https://github.com/MonoGame/MonoGame/issues/6695#issuecomment-493408960

Ejercicio propuesto 1.3.1: Descarga e instala MonoGame. Comprueba que consigues crear un proyecto usando la plantilla básica y llegar hasta esa pantalla azul.

1.4. El "bucle de juego clásico" frente al bucle de juego de MonoGame

La mayoría de los juegos se ejecutan siguiendo un bucle, con una apariencia relativamente similar a ésta:

```
repetir:
    comprobar entrada del usuario
    decidir siguientes acciones del juego (ejecutar IA)
    mover enemigos
```

```
    comprobar colisiones
    dibujar gráficos
    reproducir sonidos
hasta final de juego
```

Hay motores / bibliotecas de juegos que usan una versión más detallada y otros que usan una versión más simplificada, lo que obliga al usuario (si lo desea) a acercarse "manualmente" a ese esquema habitual. Hay algunas bibliotecas, como SDL, de nivel muy bajo, que no incluyen ningún bucle de juego preestablecido, y todo queda a discreción del programador. En el caso de MonoGame, nuestro esqueleto de juego tiene apenas dos funciones que corresponden a su "bucle de juego": Draw (que se usará para mostrar los elementos en pantalla) y Update (que se encargará de todo lo demás: comprobar entrada del usuario, mover los elementos del juego, comprobar colisiones y reproducir sonidos).

De hecho, éste es el contenido del fichero Game.cs (tras eliminar la mayoría de comentarios):

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace MonoGame02
{

    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

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

            base.Initialize();
        }

        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
        }

        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
        }
```

```
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
                        ButtonState.Pressed ||
                        Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

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

            // TODO: Add your drawing code here

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

Los métodos que podemos encontrar son:

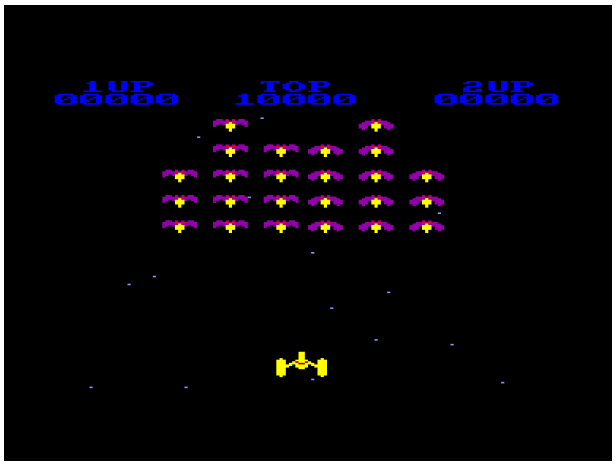
- Game1() es el constructor.
- Initialize() se usará para inicializar componentes que no sean gráficos, si es que llegásemos a tener alguno. Se llama posteriormente al constructor.
- LoadContent() cargará las imágenes, tipos de letra y sonidos, como veremos más adelante.
- UnloadContent() se podría usar para descargar recursos de memoria. No es habitual que necesitemos llamarlo, ya que el propio sistema irá liberando los recursos que ya no se usen.
- Draw(GameTime gameTime), como ya hemos anticipado, se usará para mostrar los elementos en pantalla. El parámetro GameTime (que se le pasa automáticamente) lo podremos emplear para conseguir que la velocidad sea lo más uniforme posible, ya veremos cómo.
- Update(GameTime gameTime), como también hemos adelantado, se encargará de todo lo demás relacionado con la lógica del juego: comprobar entrada del usuario, mover los elementos del juego, comprobar colisiones y reproducir sonidos.

La idea que pretende MonoGame con esta división entre "Update" y "Draw" es que en cada "frame" (fotograma, si comparamos con una animación) del juego se llame siempre a "Update", pero quizá no se llame siempre a "Draw", de modo si el ordenador no es suficientemente rápido, quizá se salte el dibujado de algún fotograma, pero la lógica se siga procesando a la máxima velocidad posible.

1.5. Cargar y mostrar un sprite

Cuando uno comienza en la programación de juegos, suele ser más recomendable imitar un juego existente, para tener un objetivo claro al que irse acercando. Si además ese juego se va a compartir, conviene que el juego que se imite no tenga copyright, o bien que nuestra versión sea suficientemente distinta como para no dar lugar a problemas legales.

Como no vamos a ver (aún) cómo añadir gravedad, voy a optar por un "matamarcianos", uno muy sencillo y del que me sea fácil **capturar sprites** usando un emulador. En mi caso, va a ser un tal "Space Hawks", de mediados de los años 80, para Amstrad CPC.



https://www.youtube.com/watch?v=0A6o3NpUVtE

A partir de una captura del juego original, debemos **redimensionar las imágenes** para la resolución que queremos que tenga nuestro juego. En mi caso, como el juego original tenía una resolución de 320x200 puntos, una alternativa razonable es redimensionarlo al 300%, para obtener un tamaño de 960x600, inferior al de una pantalla HD típica de 1366x768 píxeles, pero razonablemente cercano. Si queremos respetar la estética retro, puede ser preferible no remuestrar la imagen (Lanczos, bilineal, etc), sino elegir la opción de cambiar tamaño conservando el color del pixel más cercano (que, según el editor de imágenes que ese emplee, puede tener nombres como "mayor proximidad" o "vecino más próximo").

Una vez que tenemos la captura redimensionada a tamaño 960x600 (o el que hayamos escogido), ya podemos empezar a **recortar sprites** para nuestra nave, enemigos, disparos, etc.

Por defecto, la ventana que crea MonoGame para el juego es de tamaño **800x480**, que es una resolución un tanto baja para un juego actual. Lo habitual será que queramos adaptarla al tamaño que hemos decidido para nuestro juego, añadiendo líneas como éstas al final del constructor:

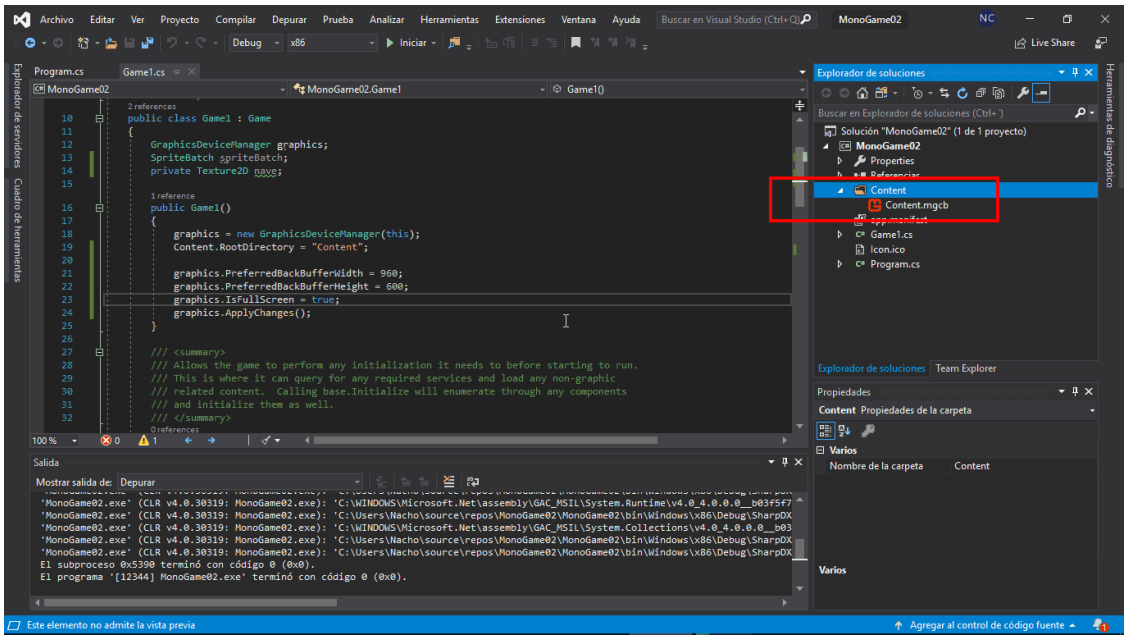
```
graphics.PreferredBackBufferWidth = 960;
graphics.PreferredBackBufferHeight = 600;
graphics.ApplyChanges();
```

Si además queremos que el juego funcione a pantalla completa, podemos añadir esta otra orden antes de "ApplyChanges":

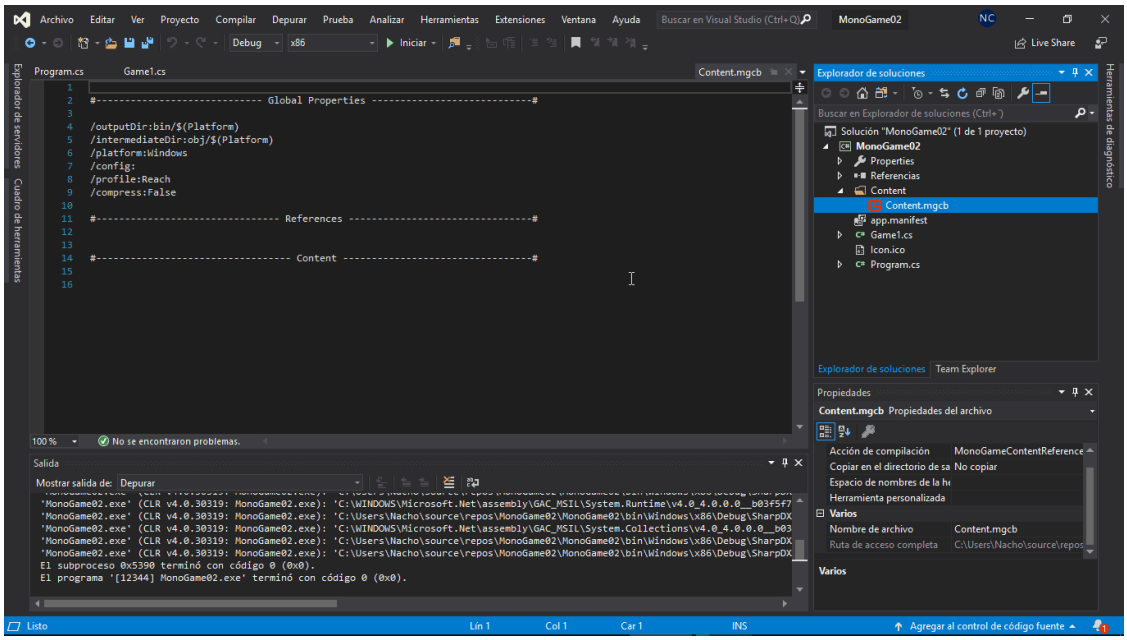
```
graphics.IsFullScreen = true;
```

La forma propuesta de incluir imágenes en XNA y, por tanto, en MonoGame, es convertirlas primero a su formato nativo, usando una herramienta llamada "**content pipeline**".

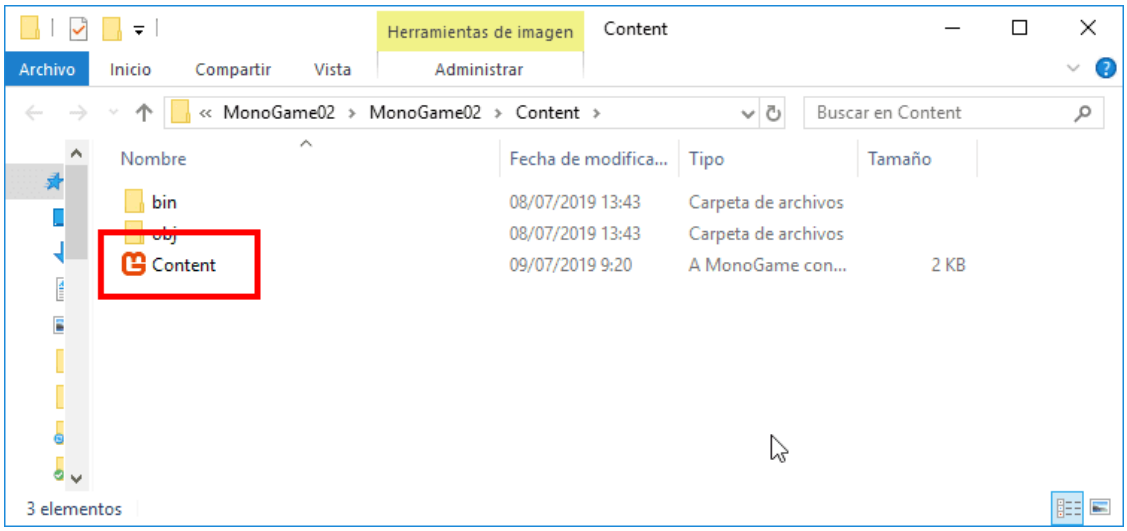
Esta herramienta aparece en el panel lateral del "Explorador de Soluciones", dentro del apartado "Content":



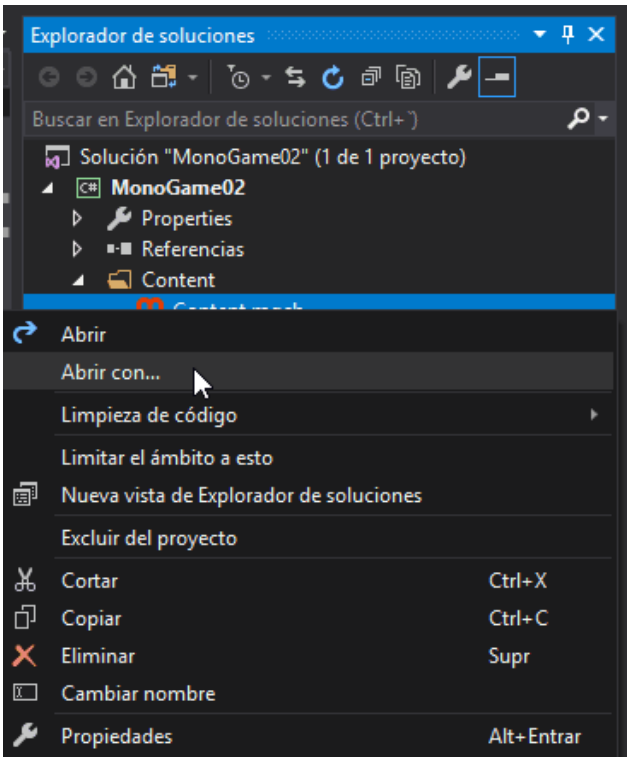
En versiones "antiguas" de Visual Studio (2015 y anteriores) esta herramienta se podía lanzar haciendo clic en ella, pero en versiones más recientes (2017 y 2019) es habitual que se nos muestre en forma de fichero de texto:



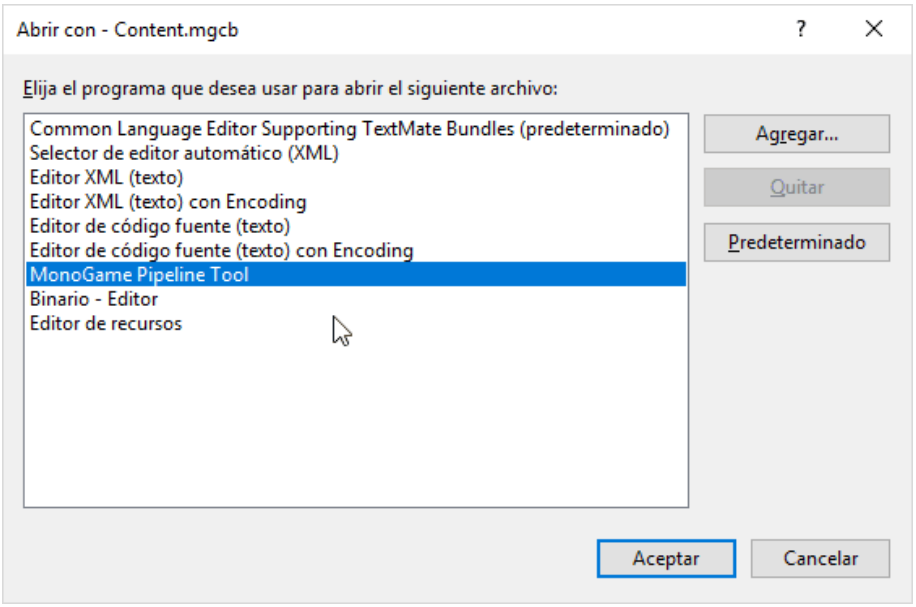
En ese caso, para lanzarla, podemos minimizar Visual Studio, ir a la carpeta "Content" de nuestro proyecto y hacer doble clic en el fichero llamado "content.mgcb":



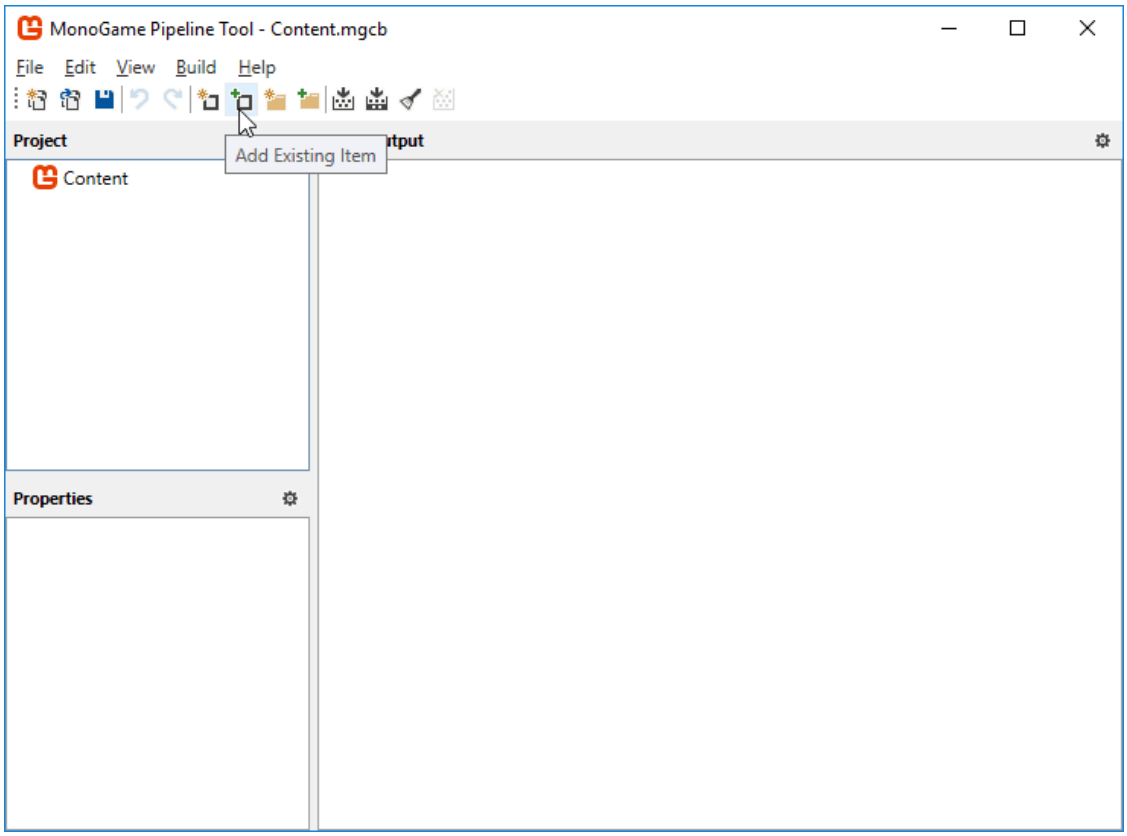
Aunque también es esperable que se pueda abrir pulsando el botón derecho (en el Explorador de soluciones) y escogiendo "Abrir con":



Y, en la lista que se nos presenta, escogiendo "MonoGame Pipeline Tool" o, en caso de que no apareciese, pulsando el botón "Agregar" y buscando dicha herramienta, que debería estar en una carpeta parecida a "C:\Program Files (x86)\ MSBuild\MonoGame\ v3.0\Tools\ Pipeline.exe":

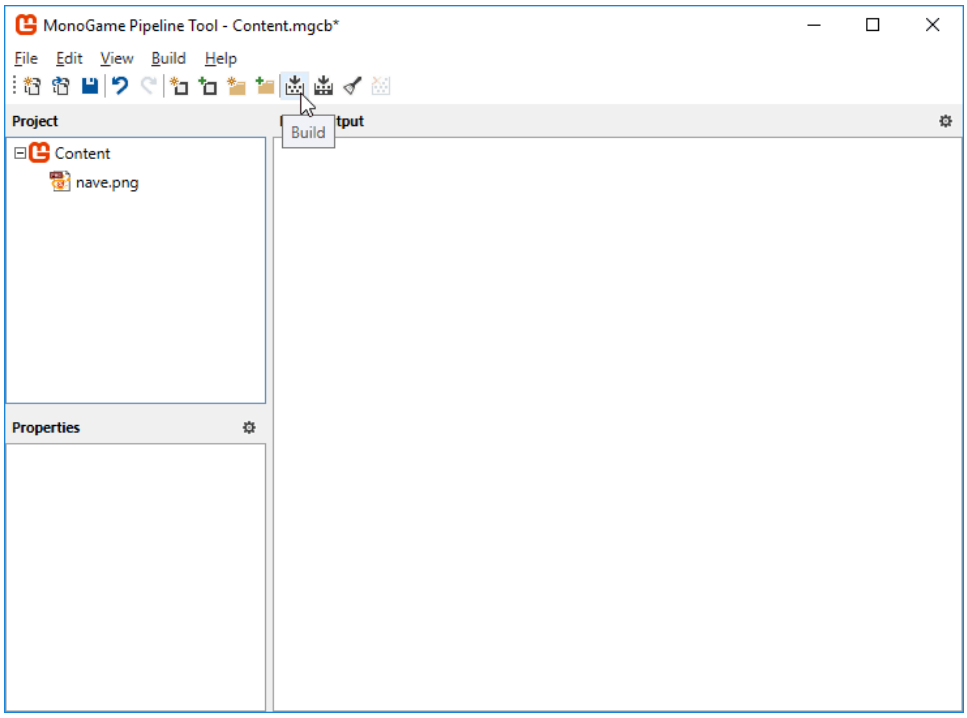


Tanto si hacemos doble clic desde la carpeta como si usamos "Abrir con", debería aparecer una ventana como ésta:

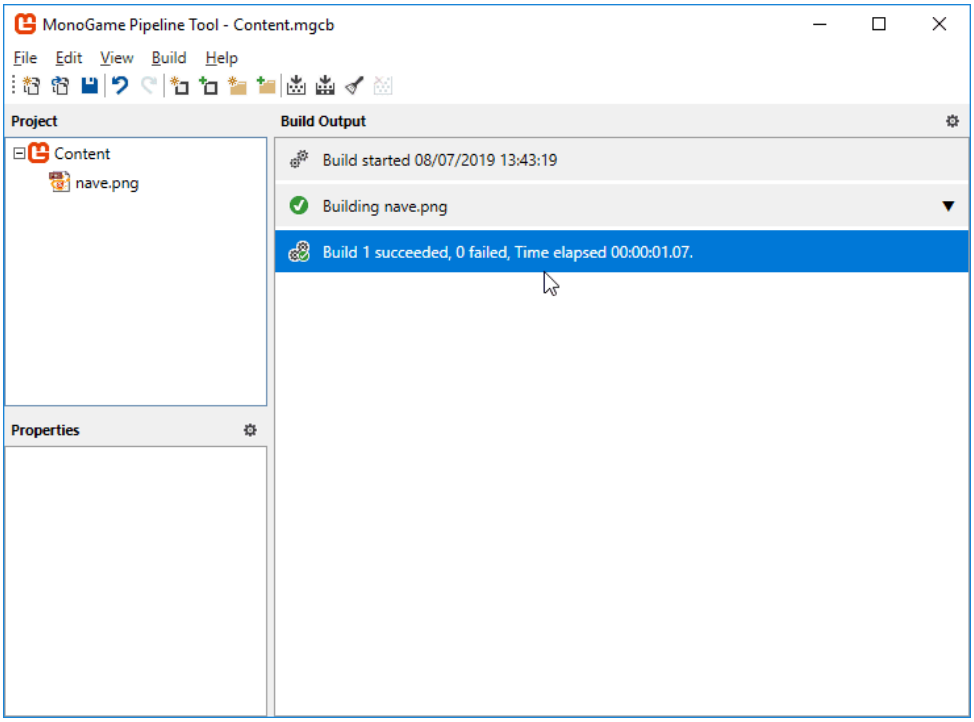


En esa ventana hay un menú botón "Add Existing Item" para añadir un elemento existente, como la imagen que habíamos preparado de nuestro sprite.

Tras añadir una o varias imágenes, podemos pulsar el botón Build para generar las imágenes en el formato de MonoGame / XNA:



Y en un instante, si todo va bien la imagen (o imágenes) estarán listas para usar:



Una vez que tenemos la imagen convertida, ya podemos añadirla al juego. Las imágenes (sprites y fondo) serán del tipo de datos "**Texture2D**", así que añadiremos un nuevo atributo:

```
private Texture2D nave;
```

Y las deberemos **cargar** al principio del juego, desde el método LoadContent, al que añadiremos la línea:

```
nave = Content.Load<Texture2D>("nave");
```

(el nombre "nave" es el mismo que el que tenía nuestro fichero inicial, pero sin la extensión ".png").

Finalmente, para **dibujar** una imagen, ésta se debe añadir a un "SpriteBatch" (lote de sprites). A la hora de dibujarla, será necesario indicar un rectángulo y un color de tintado. En cuanto al rectángulo, si queremos dibujar en las coordenadas (400,500) una nave de ancho 96 y alto 48, deberíamos usar "new Rectangle(400, 500, 96, 48)". En cuanto al color, lo habitual es usar blanco, para no alterar el color original de la imagen: "Color.White".

Por tanto, el fragmento que debemos añadir a nuestro método "Draw", antes de "base.Draw" sería algo como

```
spriteBatch.Begin();
spriteBatch.Draw(nave, new Rectangle(400, 500, 96, 48), Color.White);
spriteBatch.End();
```

De paso, podemos cambiar el color de fondo, para que en vez de ser el azul por defecto ("Color.CornflowerBlue") sea un gris oscuro definido a partir de una cierta cantidad de rojo, verde y azul (por ejemplo, "new Color(32,32,32)" si queremos un gris bastante oscuro, con apenas 1/8 de la intensidad de un blanco puro). El método Draw completo quedaría así:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear( new Color(32,32,32) );

    spriteBatch.Begin();
    spriteBatch.Draw(nave, new Rectangle(400, 500, 96, 48), Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Ejercicio propuesto 1.5.1: Prepara una imagen, conviértela a formato de MonoGame con la "content pipeline tool" y muéstrala.

1.6. Mover el sprite con el teclado

Apenas necesitamos dos detalles nuevos:

- Que las coordenadas de la nave no estén prefijadas.
- Saber comprobar el estado del teclado.

Para que las coordenadas sean variables, usaremos un nuevo atributo, de tipo "vector de 2 dimensiones":

```
private Vector2 posicionNave;
```

Y le daremos valor inicial en Initialize:

```
posicionNave = new Vector2(400, 500);
```

Para mirar si se pulsa alguna tecla, miraremos el valor de "Keyboard.GetState()" dentro del método "Update":

```
var estadoTeclado = Keyboard.GetState();
```

Ese "Keyboard.GetState()" nos permite saber si una cierta tecla está pulsada ("IsKeyDown(...)"):

```
if (estadoTeclado.IsKeyDown(Keys.Left))
    posicionNave.X -= 2;
```

Y, ahora que tenemos una X y una Y, deberemos dibujar en las coordenadas reales de la nave:

```
spriteBatch.Draw(nave,
   new Rectangle((int) posicionNave.X, (int) posicionNave.Y,
   nave.Width, nave.Height),
   Color.White);
```

Aun así, en general, será más fiable no incrementar / decrementar un tamaño fijo, sino tener en cuenta el tiempo transcurrido desde el último fotograma, por lo que la forma habitual será la siguiente:

```
if (estadoTeclado.IsKeyDown(Keys.Left))
    posicionNave.X -= velocidadNave
        * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

donde "velocidadNave" sería un nuevo atributo de tipo "float", cuyo valor daríamos en Initialize, y que podría ser algo como

```
velocidadNave = 200;
```

La idea detrás de **GameTime** es la siguiente: imaginemos que deseamos que un sprite se mueva 200 píxeles por segundo. Si nuestro juego se mueve de forma fluida a 50 fotogramas por segundos, avanzará 4 píxeles en cada fotograma. Si hay una congestión en el sistema (ya sea por cálculos de nuestro juego o por tareas externas, del sistema operativo), y baja momentáneamente a 10 fotogramas por segundo, el sprite avanzará 20 pixeles en cada fotograma, con lo que el movimiento será "menos suave" pero la velocidad efectiva del juego será la misma.

Ejercicio propuesto 1.6.1: Amplía tu esqueleto, para que la imagen que has escogido se pueda mover con las flechas del teclado.

1.7. Un segundo sprite que se mueve solo

Ya sabemos todo lo necesario para añadir "un enemigo" que se mueva solo: deberemos conseguir la imagen (por ejemplo, capturando y recortando, o descargando desde algún repositorio de Internet, o creándola nosotros mismos) y convertirla con la "pipeline tool", cargarla en LoadContent, preparar un vector de posición y cambiar su valor en "Update". Además, si queremos que se pueda mover tanto en horizontal como en vertical, podríamos usar también un Vector2 para su velocidad, en vez de un único número, de modo que su velocidad tenga una componente X (horizontal) y otra componente Y (vertical).

Con esos cambios, la clase "Game" completa podría quedar así:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace MonoGame02
{
    /// Game1: El juego en sí
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        // Datos de la nave
        private Texture2D nave;
        private Vector2 posicionNave;
        private float velocidadNave = 200;

        // Datos del enemigo
        private Texture2D enemigo;
        private Vector2 posicionEnemigo;
        private Vector2 velocidadEnemigo;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

            graphics.PreferredBackBufferWidth = 960;
            graphics.PreferredBackBufferHeight = 600;
            graphics.IsFullScreen = true;
            graphics.ApplyChanges();
        }

        /// Initialize: dar valores iniciales
        protected override void Initialize()
        {
            base.Initialize();
            posicionNave = new Vector2(400, 500);
            posicionEnemigo = new Vector2(300, 100);
            velocidadEnemigo = new Vector2(150, 100);
        }
```

```
/// LoadContent: Cargar imágenes y demás contenido
protected override void LoadContent()
{
    // Preparamos el SpriteBatch para dibujar todas las texturas
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Cargamos imágenes desde el "Content"
    nave = Content.Load<Texture2D>("nave");
    enemigo = Content.Load<Texture2D>("enemigo1a");
}

/// Update: Lógica del juego (actualizar estados, comprobar colisiones,
///     analizar entrada, reproducir audio)
/// Parámetro gameTime: para temporización
protected override void Update(GameTime gameTime)
{
    // ¿Hay que salir?
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back==ButtonState.Pressed
            || Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // Movemos nave si se pulsa teclado
    var estadoTeclado = Keyboard.GetState();
    if (estadoTeclado.IsKeyDown(Keys.Left))
        posicionNave.X -= velocidadNave
            * (float)gameTime.ElapsedGameTime.TotalSeconds;
    if (estadoTeclado.IsKeyDown(Keys.Right))
        posicionNave.X += velocidadNave
            * (float)gameTime.ElapsedGameTime.TotalSeconds;

    // Hacemos que el enemigo se mueva solo
    posicionEnemigo.X += velocidadEnemigo.X
        * (float)gameTime.ElapsedGameTime.TotalSeconds; ;
    posicionEnemigo.Y += velocidadEnemigo.Y
        * (float)gameTime.ElapsedGameTime.TotalSeconds; ;
    // TO DO: Evitar "números mágicos"
    if ((posicionEnemigo.X < 20) || (posicionEnemigo.X > 850))
        velocidadEnemigo.X = -velocidadEnemigo.X;
    if ((posicionEnemigo.Y < 20) || (posicionEnemigo.Y > 550))
        velocidadEnemigo.Y = -velocidadEnemigo.Y;

    // Y delegamos en la clase superior
    base.Update(gameTime);
}

/// Draw: dibuja elementos en pantalla
/// Parámetro gameTime: para temporización
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear( new Color(32,32,32) );

    // Dibujamos nave y enemigos
    spriteBatch.Begin();
    spriteBatch.Draw(enemigo,
       new Rectangle((int)posicionEnemigo.X, (int)posicionEnemigo.Y,
       nave.Width, nave.Height),
       Color.White);
```

```
            spriteBatch.Draw(nave,
               new Rectangle((int) posicionNave.X, (int) posicionNave.Y,
               nave.Width, nave.Height),
               Color.White);
            spriteBatch.End();

            // Y delegamos en la clase superior
            base.Draw(gameTime);
        }
    }
}
```

En "Draw", los elementos se dibujan desde el primero que indiquemos en el SpriteBatch hasta el último, de modo que **si coinciden** dos sprites en la misma posición de pantalla, se verá "por encima" el último que aparezca enumerado como parte del SpriteBatch.

Ejercicio propuesto 1.7.1: Añade a tu esqueleto una segunda imagen que se mueva sola.

1.8. Comprobación de colisiones

La forma más sencilla de comprobar colisiones es viendo si dos rectángulos se solapan. Para eso, tenemos ya preparada una clase "Rectangle", con un método "Intersects", que permite saber si se cruza con otro cierto rectángulo. Así, podríamos comprobar colisiones si añadimos al método "Update" algo como:

```
if (new Rectangle((int)posicionNave.X, (int)posicionNave.Y,
     nave.Width, nave.Height).Intersects(
        new Rectangle((int)posicionEnemigo.X, (int)posicionEnemigo.Y,
        enemigo.Width, enemigo.Height)))
    Exit();
```

Esta estructura se puede simplificar si nos creamos nuestra propia clase Sprite (o similar), y le añadimos un método "CollisionsWith()" que compruebe si colisiona con otro Sprite.

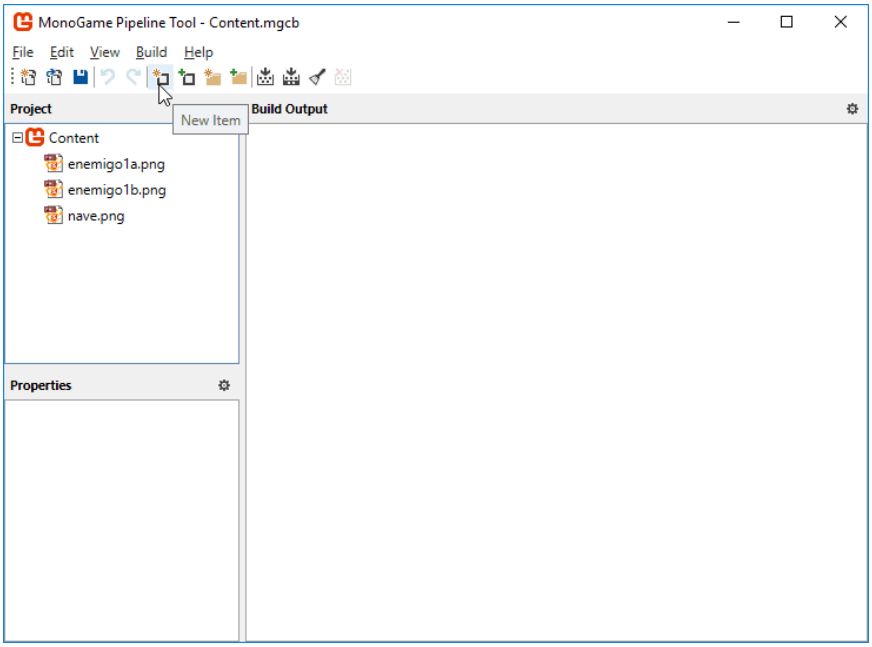
Además, en un juego real, no deberíamos terminar a la primera colisión, sino usar booleanos auxiliares que permitiesen mostrar una explosión, perder una vida, etc.

Ejercicio propuesto 1.8.1: Añade comprobación de colisiones a tu esqueleto de juego.

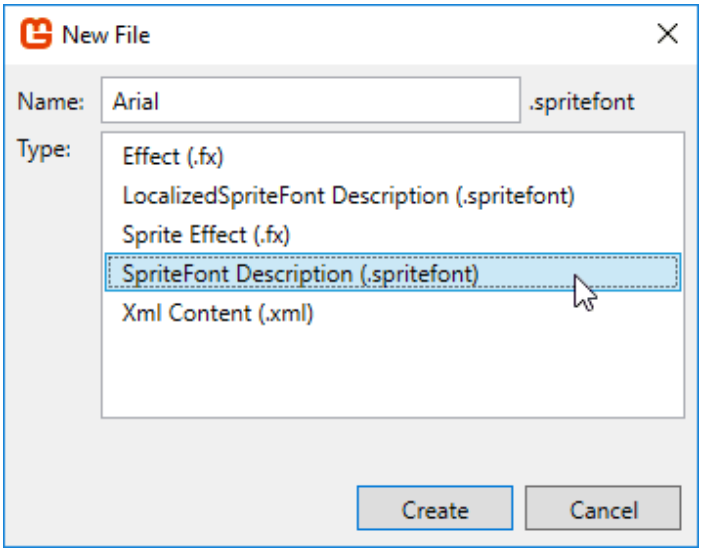
1.9. Incluir textos

La forma más sencilla de incluir textos es añadiendo al proyecto una fuente (tipo de letra) en formato TTF. Aun así, esto no es trivial en MonoGame. Los pasos que deberemos seguir son:

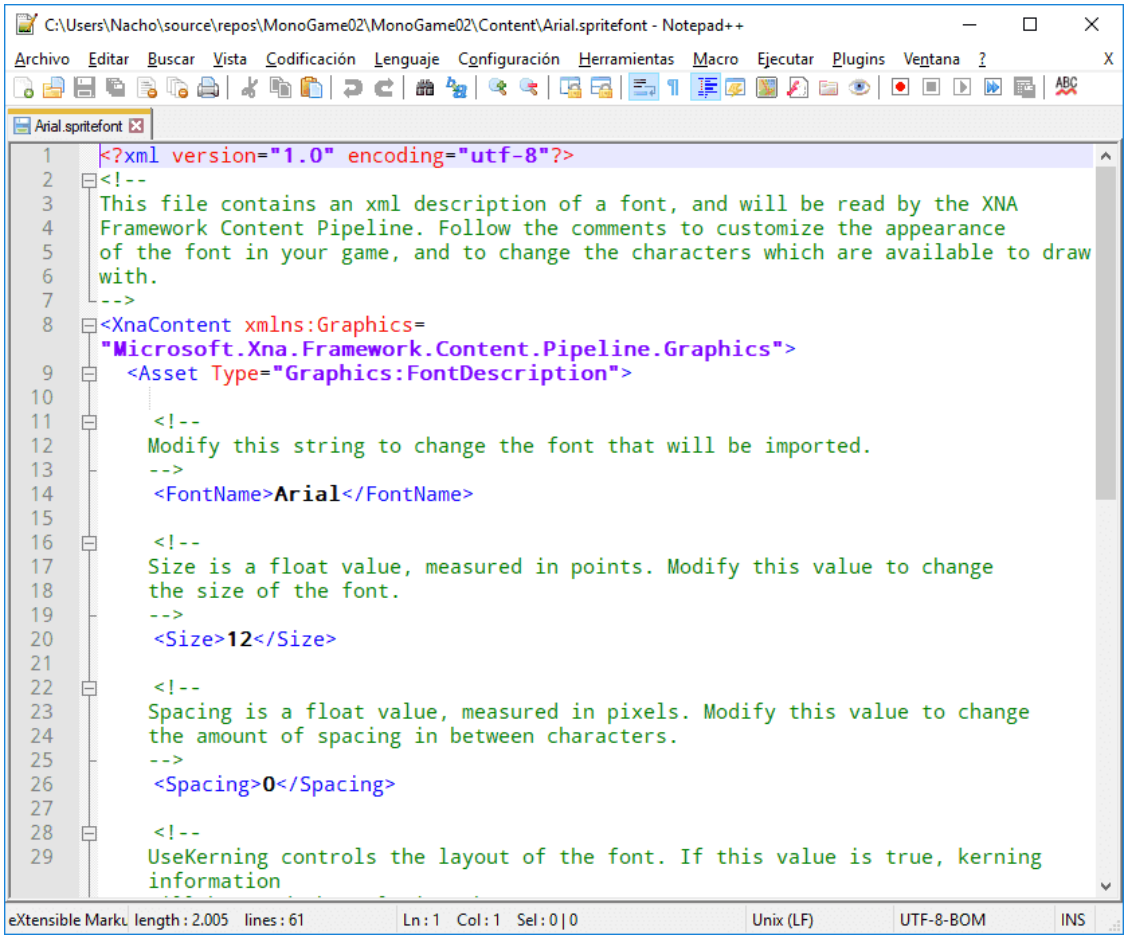
En el gestor de contenidos, esta vez no usaremos la opción de Añadir un elemento existente, sino la de Crear un nuevo elemento ("New Item").



De entre los tipos de ficheros que se nos proponen, escogeremos una "Descripción de SpriteFont" ("SpriteFont Description") y escogeremos un nombre adecuado, como podría ser "Arial".



Ahora hay que editar ese fichero, y eso no se puede hacer desde el propio gestor de contenidos. Como es un fichero XML, podremos usar cualquier editor de texto, como Notepad++ o Geany:



En ese fichero hay detalles como el nombre de la fuente, que en principio se habrá tomado de lo que hemos tecleado, o el tamaño (por ejemplo, 12 puntos), o el juego de caracteres. Este juego de caracteres, en principio, incluirá todos los del alfabeto inglés: del ASCII 32 -espacio- al ASCII 126 -~-, pero por defecto no incluirá acentos ni eñe. También se puede reducir ese rango si queremos (por ejemplo) usar sólo las cifras numéricas, y así los datos del programa ocuparían un poco menos.

Finalmente, deberemos pulsar el botón "Build" para generar los ficheros de contenido actualizados.

Ahora ya podemos usar esta fuente en nuestro programa. En primer lugar añadiremos un nuevo atributo:

```
private SpriteFont fuente;
```

Lo cargaremos desde "LoadContent":

```
fuente = Content.Load<SpriteFont>("Arial");
```

Y lo dibujamos como parte del "spriteBatch", en "Draw":

```
spriteBatch.DrawString(fuente, "Hola", new Vector2(50, 10), Color.Yellow);
```

Obviamente, en un juego real no mostraríamos el texto "Hola", sino el marcador con la puntuación, el número de vidas, el nivel actual, etc.

Un último detalle: lo ideal es no usar fuentes "comerciales", como la Arial que viene con Windows y que hemos empleado en este ejemplo, sino buscar una que sea de libre distribución y que tenga una estética adecuada para nuestro juego. Puedes encontrar muchos repositorios de fuentes en Internet, como:

https://www.dafont.com/es/

https://www.1001freefonts.com/es/

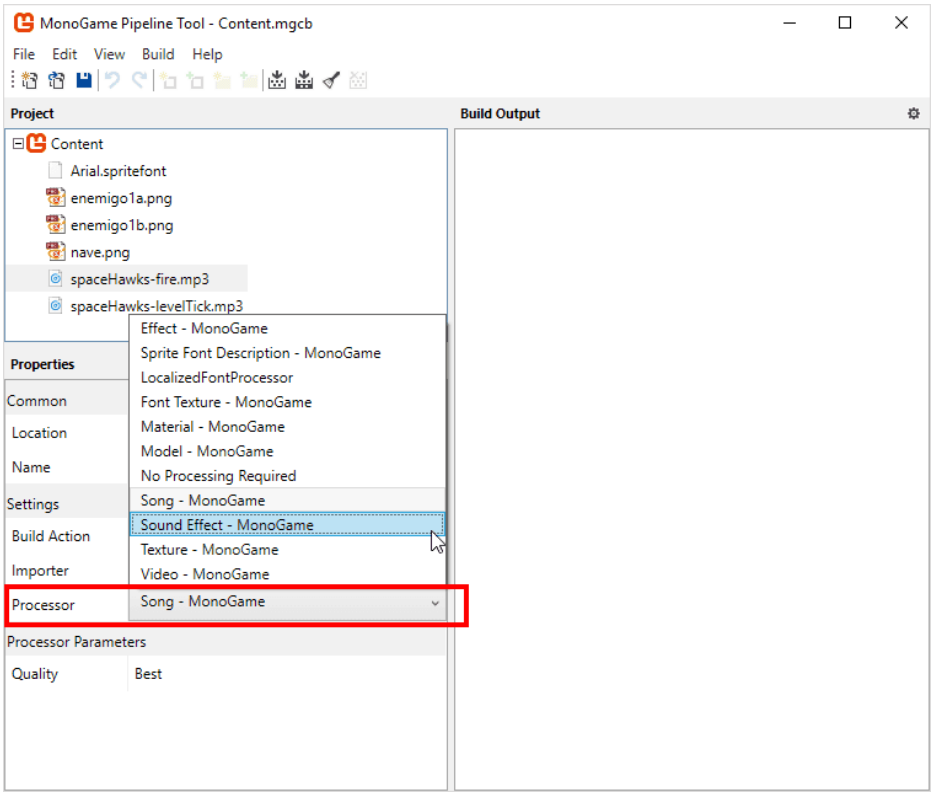
https://fonts.google.com/

Ejercicio propuesto 1.9.1: Añade un texto a tu juego, por ejemplo un (falso) marcador de puntuación.

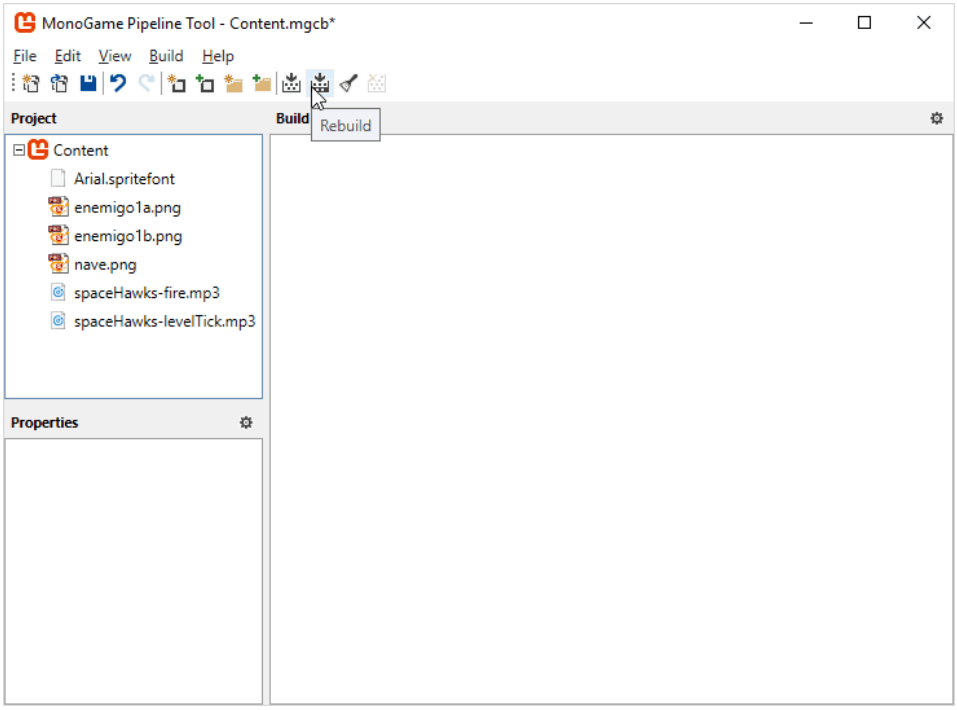
1.10. Sonido en MonoGame

Añadir sonidos es muy similar a añadir imágenes:

- Primero deberemos convertirlo a su formato nativo usando la "Pipeline tool": pulsar el botón "Añadir elemento existente" e incluir los sonidos en formato MP3 u OGG.
- Existe un paso adicional que no existe en las imágenes: puede ocurrir que todos los sonidos se tomen como "música" ("Song"), pero es habitual que para los "efectos" de sonido (como disparos o explosiones) nos interese poder reproducir varios de ellos a la vez. En ese caso, deberemos hacer clic en el panel de "Propiedades" ("Properties") y cambiar el "procesador" ("Processor") para que no sea "Song" sino "Sound Effect":



- Finalmente pulsaremos el botón de Reconstruir ("Rebuild"):



- En el fuente, posiblemente deberemos añadir un par de nuevos "using": "Media" para reproducir músicas (sólo una a la vez, quizá de forma repetitiva) y "Audio" para efectos de sonido (que sí permite varios simultáneos):

```
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Audio;
```

- Declararemos las variables que vayamos a usar:

```
private Song musicaDeFondo;
private SoundEffect sonidoDeDisparo;
```

- Después deberemos cargarlos desde "LoadContent":

```
sonidoDeDisparo = Content.Load<SoundEffect>("spaceHawks-fire");
musicaDeFondo = Content.Load<Song>("spaceHawks-levelTick");
```

- Si la música de fondo queremos que comience al principio del juego, la podríamos poner en marcha desde el propio "LoadContent", después de cargarla, así como indicar (si lo deseamos) que se repita indefinidamente:

```
MediaPlayer.Play(musicaDeFondo);
MediaPlayer.IsRepeating = true;
```

- Los efecto de sonido se reproducirían desde "Update", llamando al método "Play()" y, preferiblemente, creando una instancia antes de hacerlo, para permitir varios sonidos simultáneos del mismo tipo. Por ejemplo, nuestro esqueleto de juego podría reproducir el sonido de disparo (aunque realmente aún no dispara) cuando se pulse la tecla Espacio:

```
if (estadoTeclado.IsKeyDown(Keys.Space))
    sonidoDeDisparo.CreateInstance().Play();
```

Como siempre, el autocompletado de Visual Studio nos puede ayudar a investigar en caso de necesitar funcionalidades adicionales. Por ejemplo:

- Una canción se detiene con ".Stop()"
- Se puede cambiar el volumen de una canción con la propiedad "MediaPlayer.Volume" (un número real, de 0.0 a 1.0).
- Se puede saber su duración con "cancion.Duration" (que es de tipo TimeSpan) y la posición actual de la reproducción con "MediaPlayer.Position" (también es un TimeSpan).
- Se puede cambiar el volumen de los efectos con la propiedad "SoundEffect.MasterVolume" (también un número real, de 0.0 a 1.0).

Ejercicio propuesto 1.10.1: Añade una música y un efecto de sonido a tu esqueleto de juego.

1.11. Pantalla de bienvenida y de créditos

En MonoGame hay un único bucle que se repite continuamente. Eso supone que si queremos tener varias "pantallas con distinta lógica" (pantalla de presentación, pantalla de créditos, etc.) debamos llevar control "de forma artesanal" de en qué pantalla estamos.

Una forma relativamente sencilla de conseguirlo es crear varias clases, cada una de ellas con su propio Update y su propio Draw, y que la clase principal pase a ser un "GestorDePantallas", con un comportamiento relativamente parecido a éste:

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed
            || Keyboard.GetState().IsKeyDown(Keys.Escape))
    {
        Exit();
    }

    switch(modoActual)
    {
        case MODO.JUEGO: juego.Update(gameTime); break;
        case MODO.BIENVENIDA: bienvenida.Update(gameTime); break;
        case MODO.CREDITOS: creditos.Update(gameTime); break;
    }

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);
    spriteBatch.Begin();
    switch (modoActual)
    {
        case MODO.JUEGO: juego.Draw(gameTime, spriteBatch); break;
        case MODO.BIENVENIDA: bienvenida.Draw(gameTime, spriteBatch); break;
        case case MODO.CREDITOS: creditos.Draw(gameTime, spriteBatch); break;
    }
    spriteBatch.End();
    base.Draw(gameTime);
}
```

Ejercicio propuesto 1.11.1: Crea una pantalla de bienvenida, que actúe como menú para entrar al juego o para ver una pantalla de créditos, que también deberás crear.

1.12. ¿Por dónde seguir? ¿Dónde saber más?

Nuestro programa empieza a crecer. El siguiente paso, antes de añadir nuevas funcionalidades, debería ser refactorizarlo, dividirlo en clases: una clase Sprite, una clase Nave, otra clase Enemigo. A partir de ahí ya podríamos crear una lista de enemigos, un disparo, varias olas de enemigos, cada una asociada a un nivel, etc.

Pero MonoGame no es nuestro "objetivo final", sino una herramienta para conocer lo que suelen aportar las bibliotecas de juegos "convencionales" y poder comparar con Unity, que será lo que comenzaremos a utilizar en el tema 2. Por eso, no vamos a profundizar más. Si quieres investigar más por tu cuenta:

La documentación oficial de MonoGame está en:

http://www.monogame.net/documentation/?page=main

En concreto, el apartado de tutoriales está en:

http://www.monogame.net/documentation/?page=Tutorials

También puedes ver el "tutorial oficial" de XNA 4 (del año 2011) en:

https://docs.microsoft.com/en-us/previous-versions/windows/xna/bb203893%28v%3dxnagamestudio.41%29

Hay bibliotecas de sonidos en Internet, en las que puedes encontrar efectos de sonido, como:

https://downloads.khinsider.com/game-soundtracks/album/space-invaders-extreme-pc-gamerip

Ejercicio propuesto 1.12.1: Crea una animación que muestre copos de nieve cayendo. Ajusta la velocidad de caída vertical y añade algo de aleatoriedad en el movimiento horizontal para que resulte creíble.