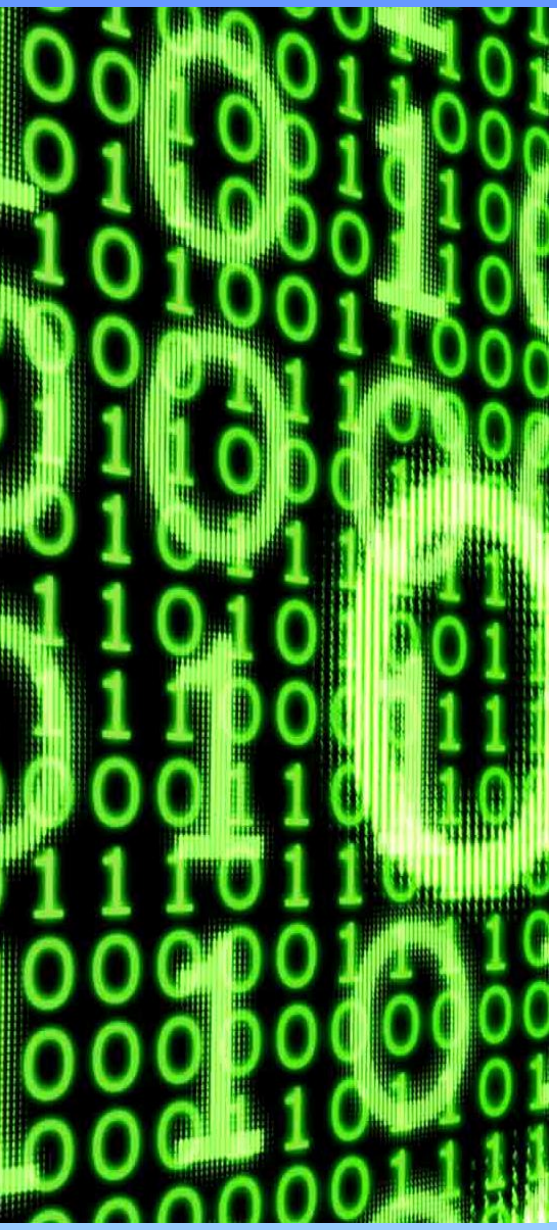


Programación de Juegos

Tema 2: Programación de Videojuegos: Arquitectura, Bibliotecas y Motores

Universidad de Huelva

Departamento de Tecnologías de la Información

Área de Ciencias de la Computación e Inteligencia Artificial

Objetivos:

- Conocer el esquema principal y los **elementos arquitectónicos** genéricos de un videojuego, con sus diferentes componentes, y las ventajas que supone trabajar partiendo de estos conceptos.

- Familiarizarse con la **secuencia lógica** de un programa de entretenimiento a nivel genérico, sin concretar el recurso de programación empleado.

- Conocer preliminarmente los recursos actuales para la programación de videojuegos.

Índice:

Índice:

1. Arquitectura

Qué…

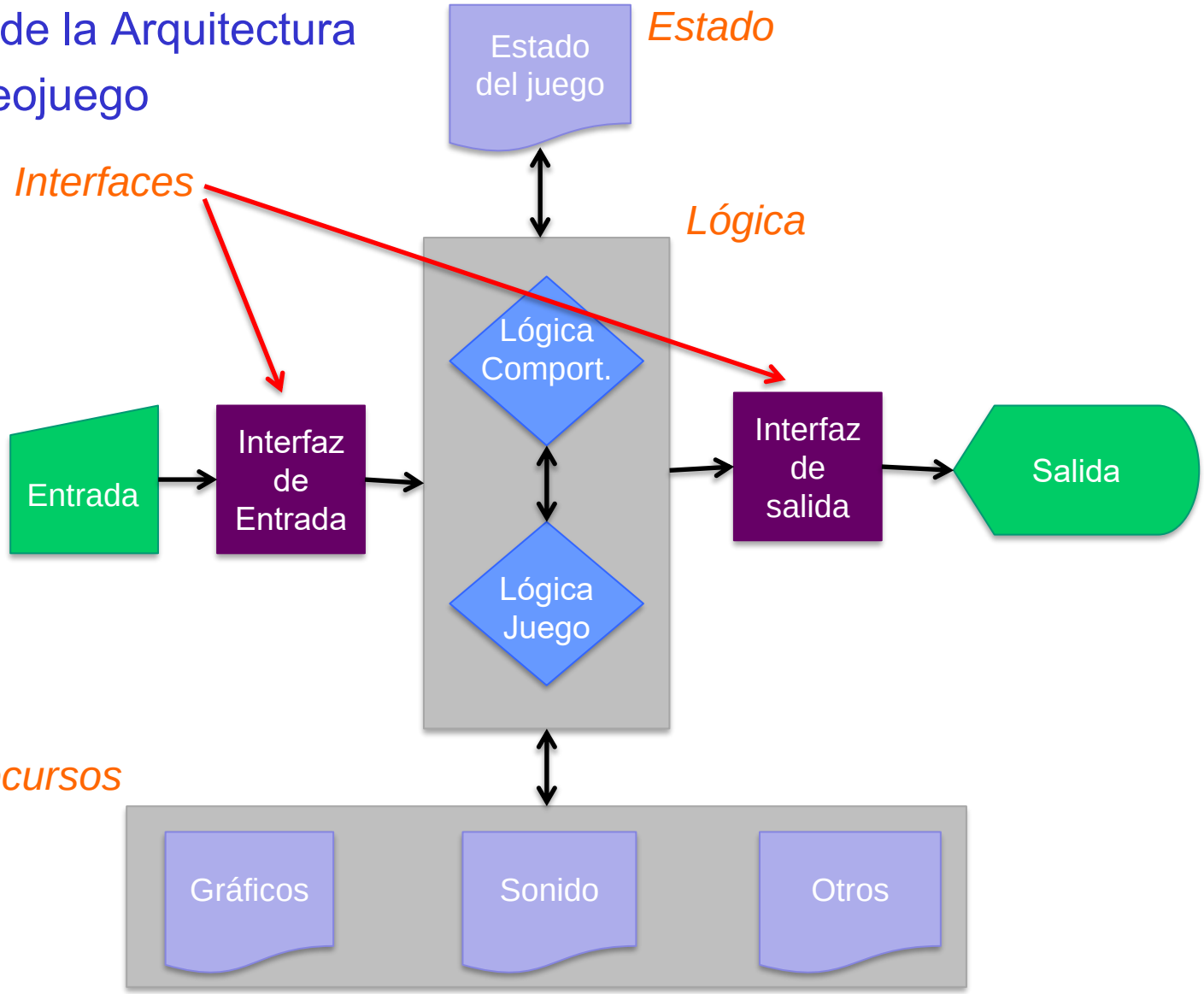
- La arquitectura de cualquier programa, es la definición abstracta de una serie de bloques o componentes que realizan alguna función dentro del mismo.

Por qué…

- Disponer de estos elementos abstractos permite un buen diseño, la reusabilidad de elementos, y la organización y especialización en el desarrollo (que redunda en mejoras de calidad, tiempo y costes).

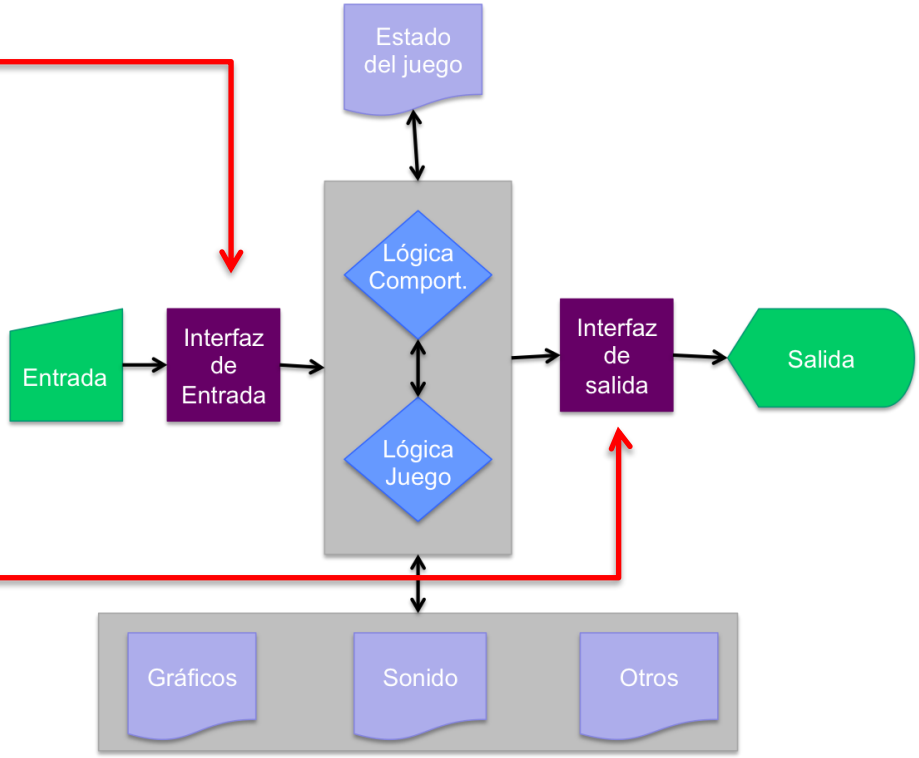
- Históricamente, el concepto de arquitectura en los videojuegos no fue un elemento que partiera desde los inicios sino que se ha ido desarrollando y adaptando posteriormente. Actualmente es imprescindible. Bibliotecas de componentes.

Esquema de la Arquitectura de un Videojuego



Interfaces:

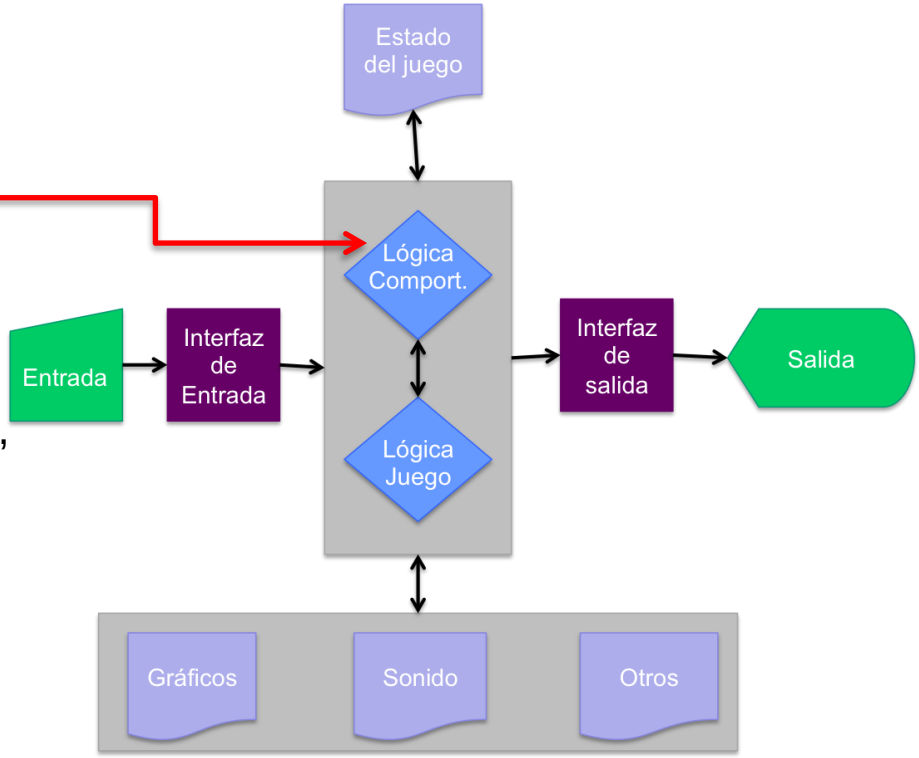
- Entrada:
  - Teclado
  - Ratón
  - *Touchpad*
  - *Joystick*
  - Mando o Controlador
  - Etc.
- Salida:
  - Pantalla/s (Gafas RV)
  - Audio
  - *Feedback* del controlador



- En un mismo videojuego, estos elementos son específicos para cada plataforma o cada sistema operativo dentro del PC.

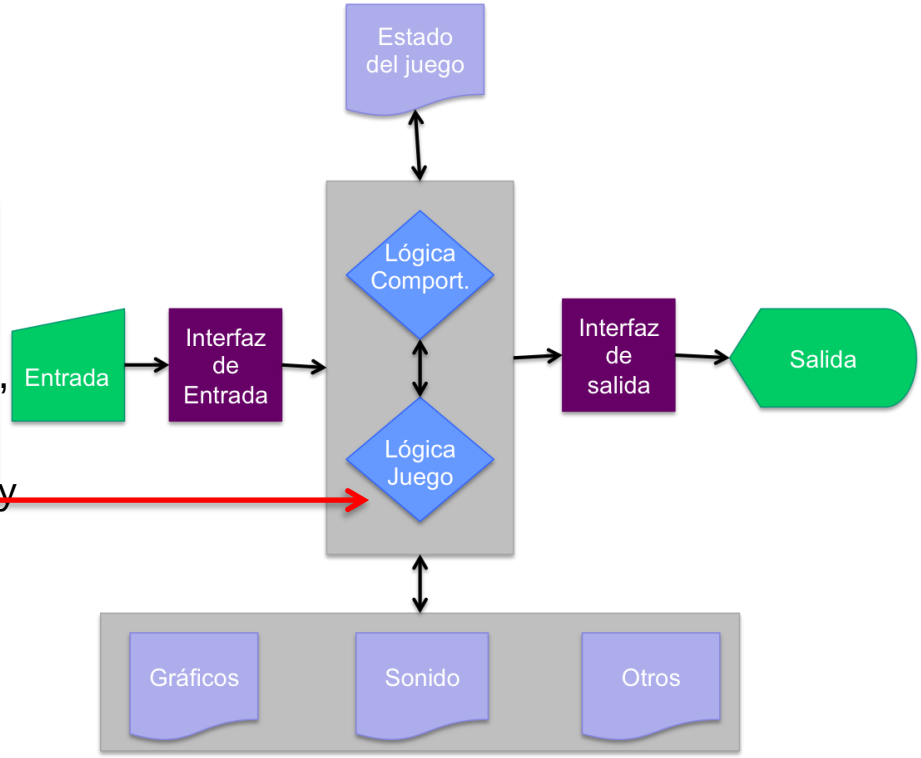
Lógica:

- Es el propio juego o “partida”
- Lógica de **comportamiento**:
  - Mantiene actualizados los elementos del programa según los principios del juego y las acciones del jugador (colisiones, IA, etc.).
    - Variables del jugador.
    - Etc., dependientes del jugador.



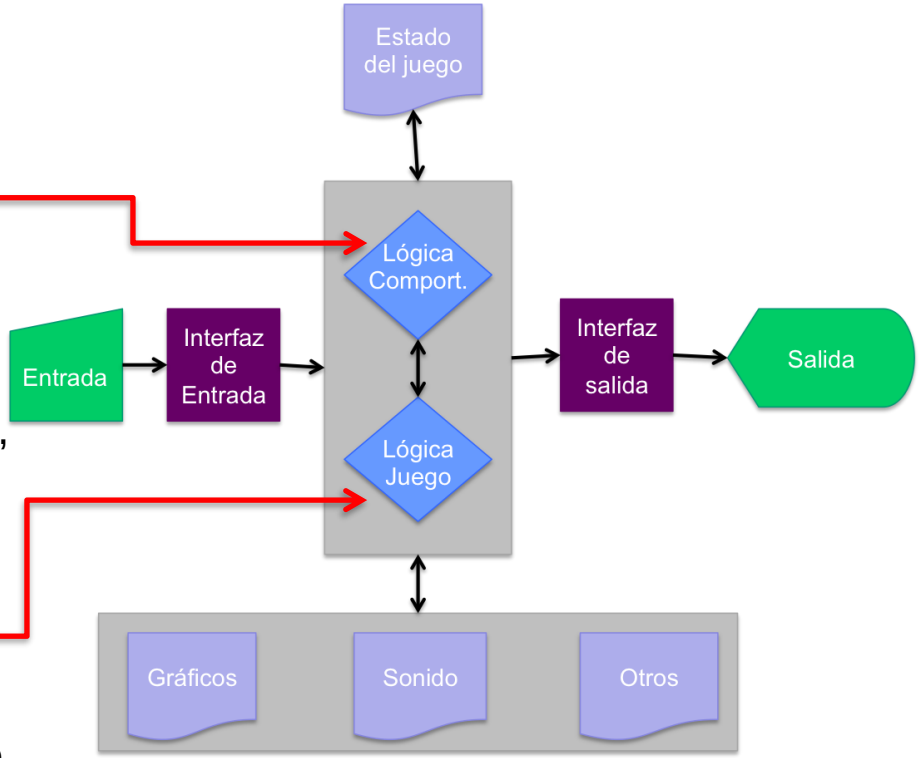
Lógica:

- Es el propio juego o “partida”
- Lógica **del juego**:
  - Organiza el resto de elementos lógicos que son independientes del jugador, tales como gráficos, E/S, sonido, etc.
    - Gráficos y Física del escenario y de los rivales
    - Alimentación de los buffers de sonido.
    - Etc., independientes de la actividad del jugador



Lógica:

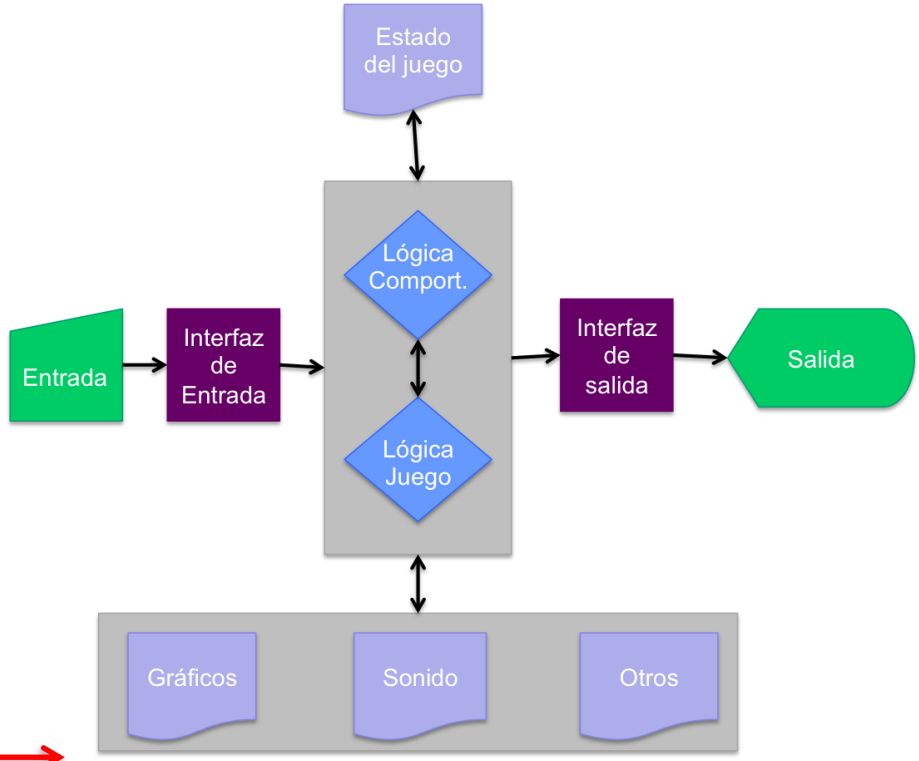
- Es el propio juego o “partida”
- Lógica de comportamiento:
  - Mantiene actualizados los elementos del programa según los principios del juego y las acciones del jugador (colisiones, IA, etc).
- Lógica del juego:
  - Organiza el resto de elementos tales como gráficos, E/S, sonido



✧ Diferencia: En la lógica del juego están todas aquellas acciones que deben ser automáticas, es decir, que no dependen directamente de la acción del jugador, mientras que la lógica de comportamiento sí lo es.

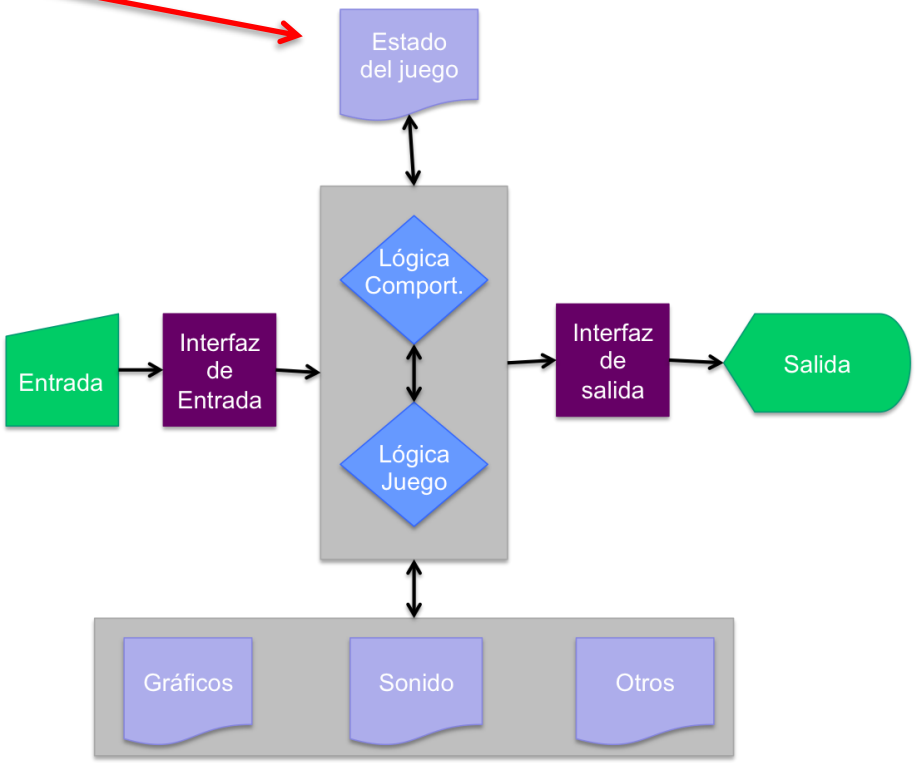
Recursos:

- Gráficos:
  - Manejo de texturas, *sprites*, animaciones, etc.
- Sonido:
  - Efectos, música, etc.
- Otros ficheros de datos:
  - Planos, mapas, configuración, partidas guardadas, etc.



Estado del juego:

- Estructuras de datos y variables que almacenan el estado instantáneo del juego
  - Variables de estado de los diferentes algoritmos (variables del juego, no del jugador).
  - Estado de los autómatas finitos



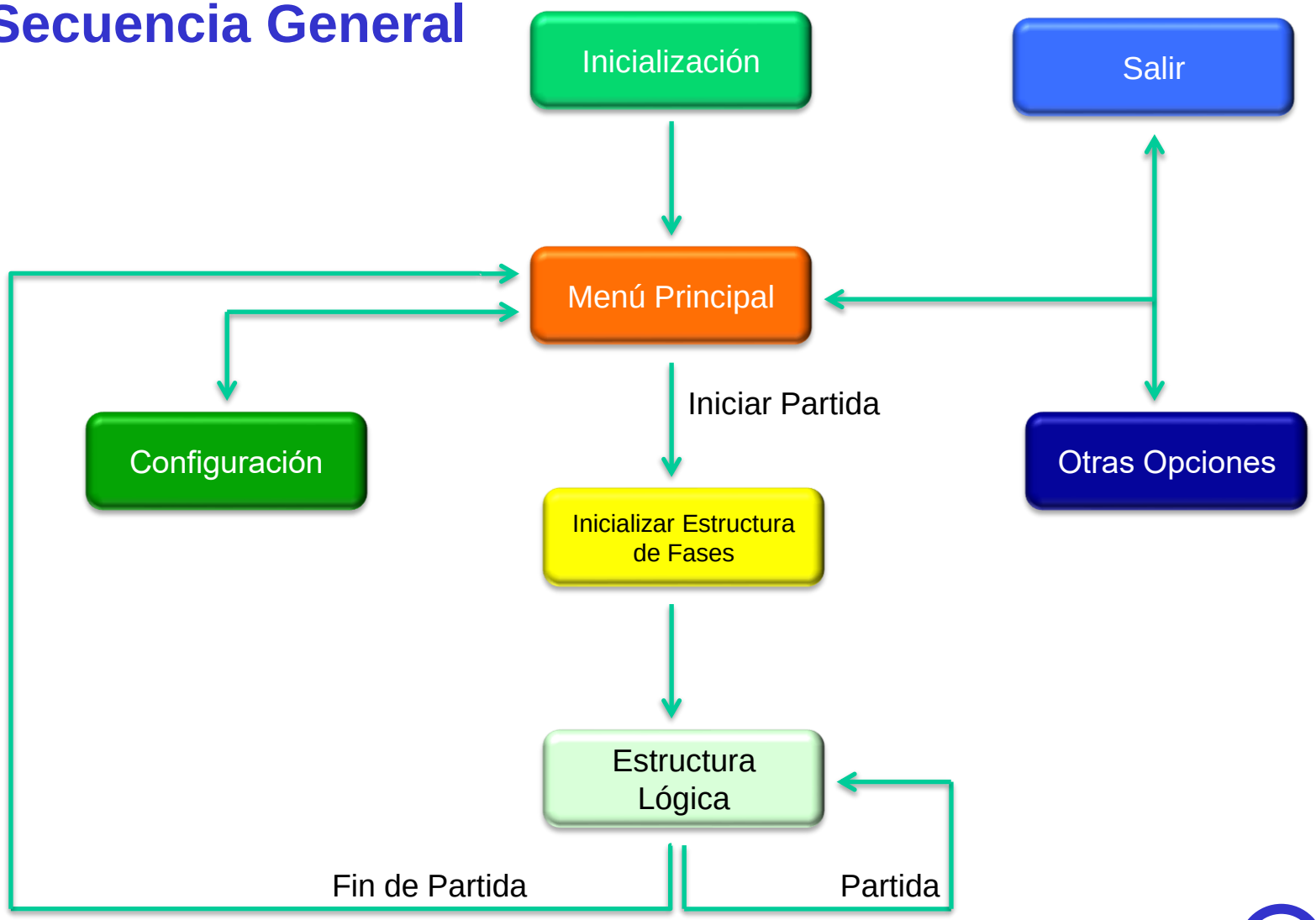
- Actividad presencial entregable:

Diseñemos nuestro propio videojuego.

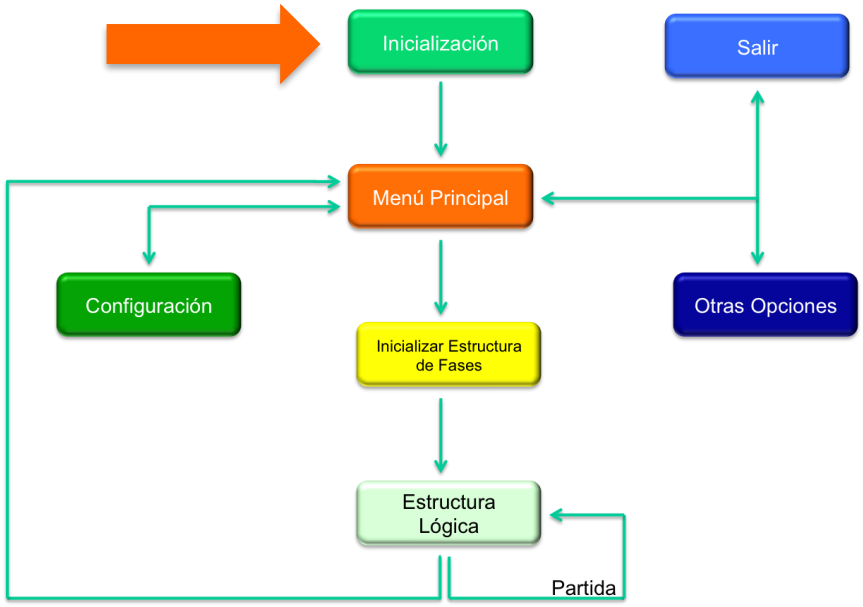
Para ello, comencemos definiendo los elementos que forman parte de cada uno de los componentes de la arquitectura del **videojuego**.

Índice:

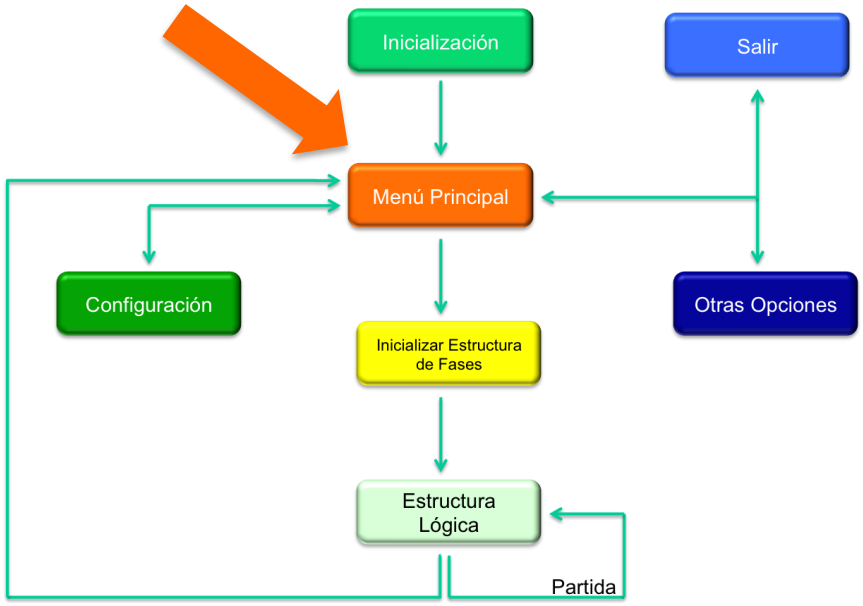
2. Secuencia General



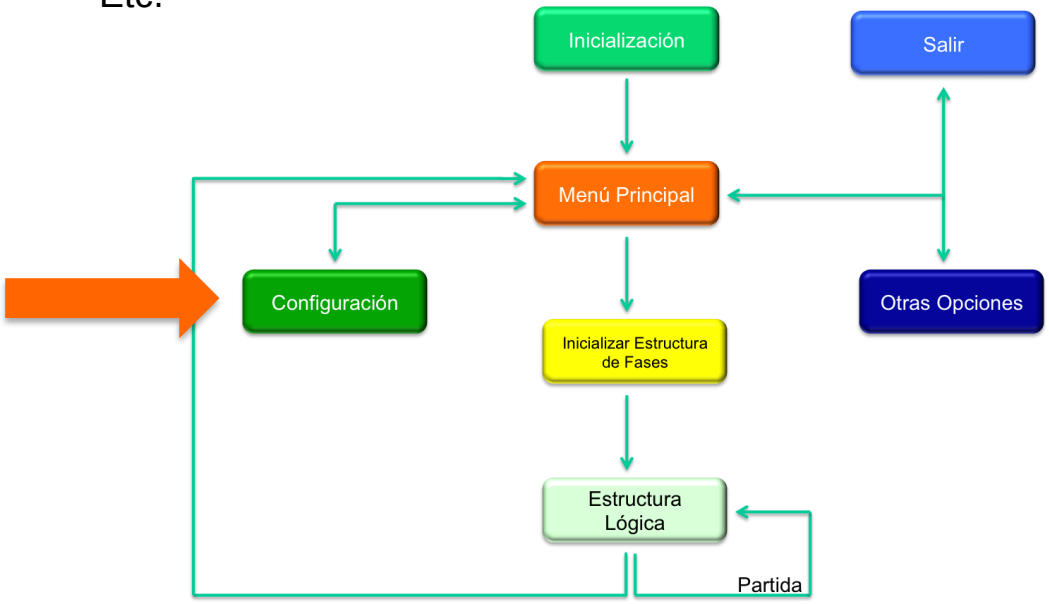
- Inicialización:
  - Cargar drivers gráficos
  - Cargar drivers de sonido
  - Chequear resto los dispositivos de E/S
  - Comprobar integridad de ficheros y datos: Inicializar/Cargar
  - Inicializar estructuras de datos
  - Iniciar el modo gráfico
  - Iniciar la secuencia de presentación-introducción



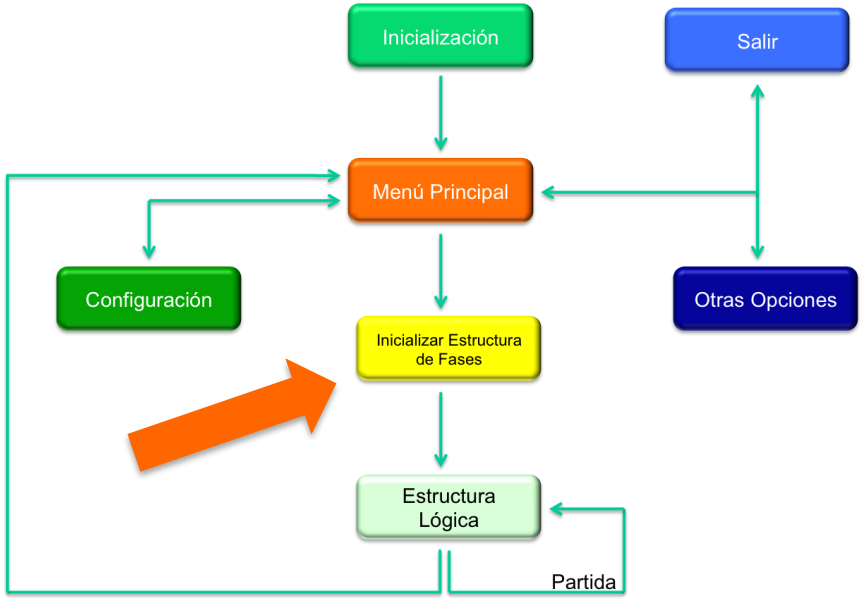
- Menú Principal (Menú del Juego):
  - Menú general del juego (partida nueva, empezar juego, cargar, salir…)
  - Lanzar submenús de configuración, otras opciones, …
  - Llevar a cabo la conectividad
  - Recuperación de partidas
  - Lanzar la secuencia de inicio de juego (en alguno de sus modos: *arcade* o no…)



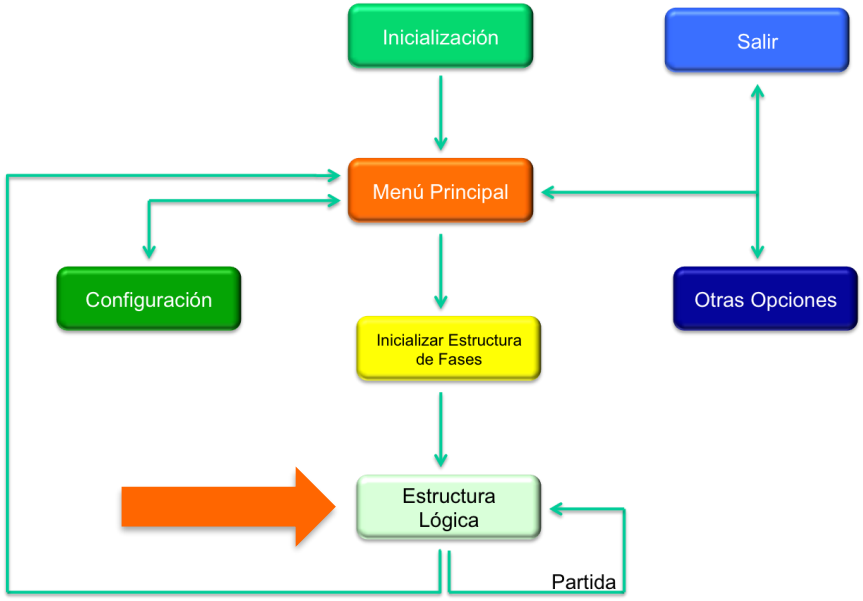
- Configuración y Otras Opciones:
  - Dependiendo del juego concreto:
    - Audio
    - Conectividad (Ajustes)
    - Opciones de video
    - Opciones de mandos
    - Gestor de partidas
    - Editores de niveles (si los hubiera), talleres, configuración de personajes/coches, y demás…
    - Etc.



- Inicializar Estructura de Fases:
  - Inicialización de variables de estado de partida
  - Carga de escenarios o datos de ficheros del nivel
  - Lanzar la secuencia del juego o partida



- Estructura / Secuencia Lógica:
  - Bucle de ejecución de la acción del juego o partida propiamente… que veremos en la siguiente sección dedicada a ella específicamente.



- Actividad presencial entregable:

Continuemos el diseño de nuestro propio videojuego.

Para ello, definiremos las acciones realizadas en cada una de las etapas de la secuencia general del **videojuego**.

Índice:

3. Secuencia Lógica

- Bucle que encadena las acciones que llevan a cabo la dinámica del juego: es el corazón del juego
- Es una especie de "en cada *frame*": todo lo que se hace entre cada imagen
- Está compuesta por tanto, por otras acciones, cuyo orden puede ser el mostrado u otro similar ya que no es completamente determinante


```
// a simple game loop in C++

int main( int argc, char* argv[] )
{
    game our_game;
    while ( our_game.is_running())
    {
        our_game.update();
    }
    return our_game.exit_code();
}
```

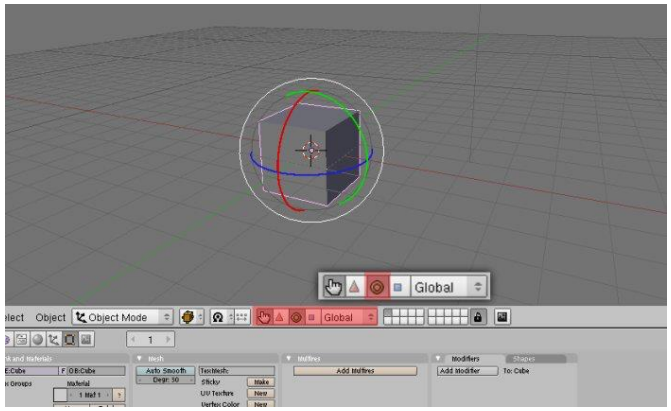
```
while( user doesn't exit )
  check for user input
  run AI
  move enemies
  resolve collisions
  draw graphics
  play sounds
end while
```

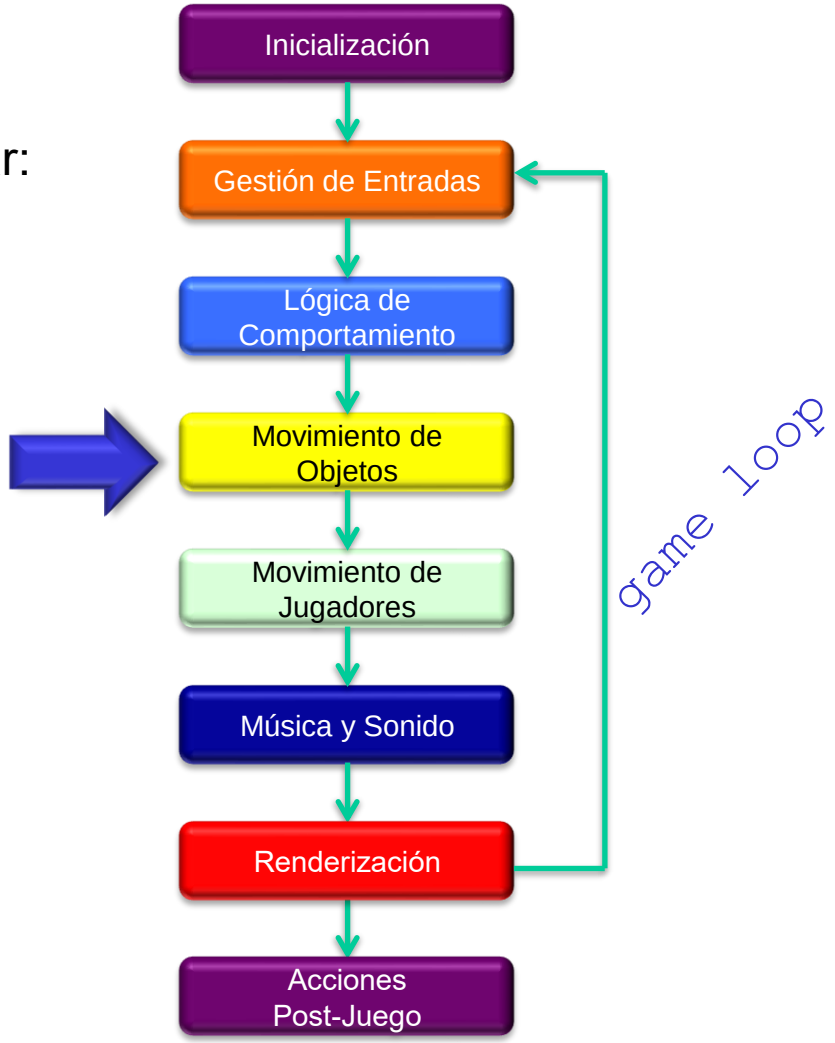
- Inicialización:
  - Reserva de memoria para las estructuras de datos dinámicas temporales de la secuencia lógica
  - Inicialización de variables del juego
  - Carga en las estructuras de datos de la información necesaria proveniente de ficheros relativa a estado de algoritmos de IA, imágenes, videos, sonidos, datos para el escenario o dinámica de la fase, etc.
  - Inicialización de variables de cada jugador (en su caso)

- Gestión de entradas (de usuario):
  - Procesar entradas de:
    - Teclado
    - Ratón
    - Mando o Controlador
    - *Joysticks* (volantes, etc).

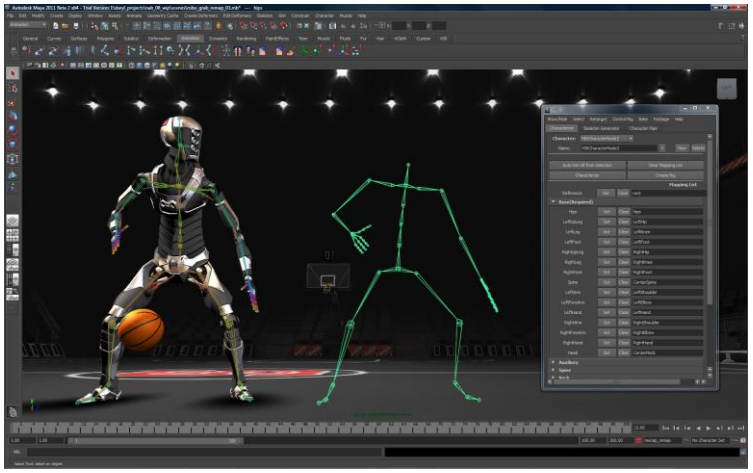
- Lógica de Comportamiento:
  - Detección de colisiones
  - Actualización de variables del juego y de los jugadores
  - Inteligencia Artificial de adversarios

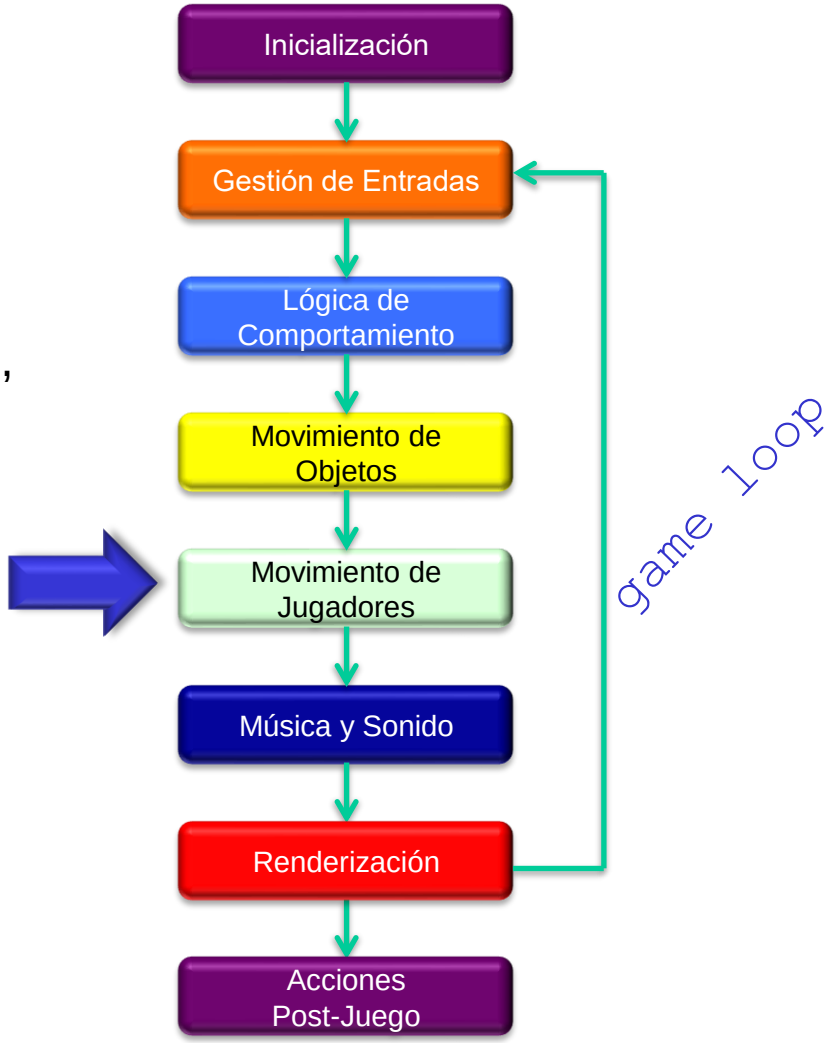
- Movimiento de Objetos:
  - Desplazar, rotar, escalar y animar:
    - objetos animados del entorno
    - adversarios (computarizados)



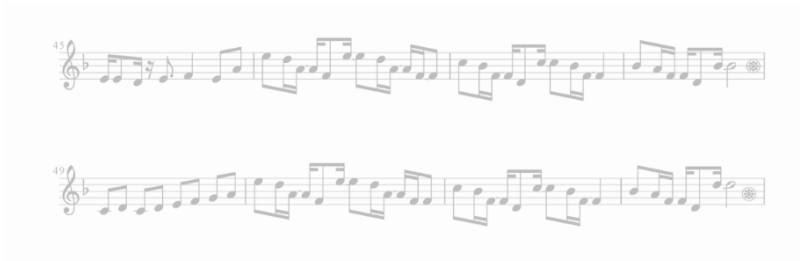


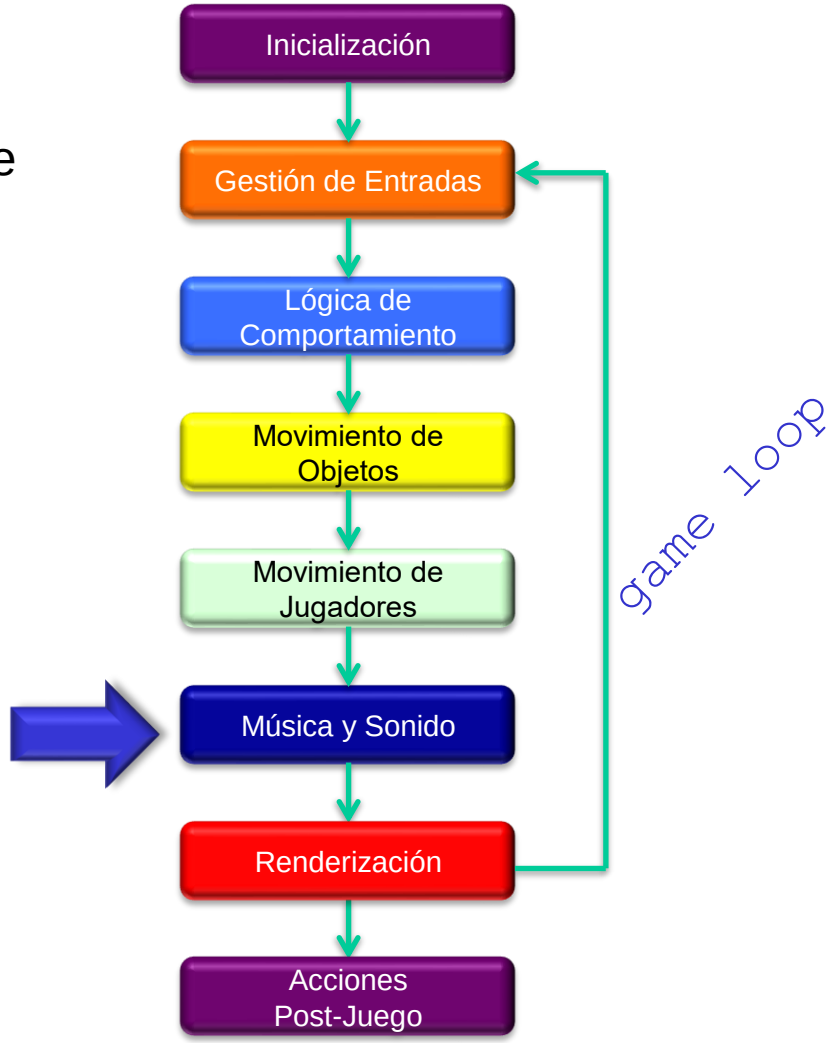
- Movimiento de Jugadores:
  - Procesar las variables de cada jugador (estado, vidas, energía, etc)
  - Actualizar movimiento: desplazar, rotar, escalar, animar…



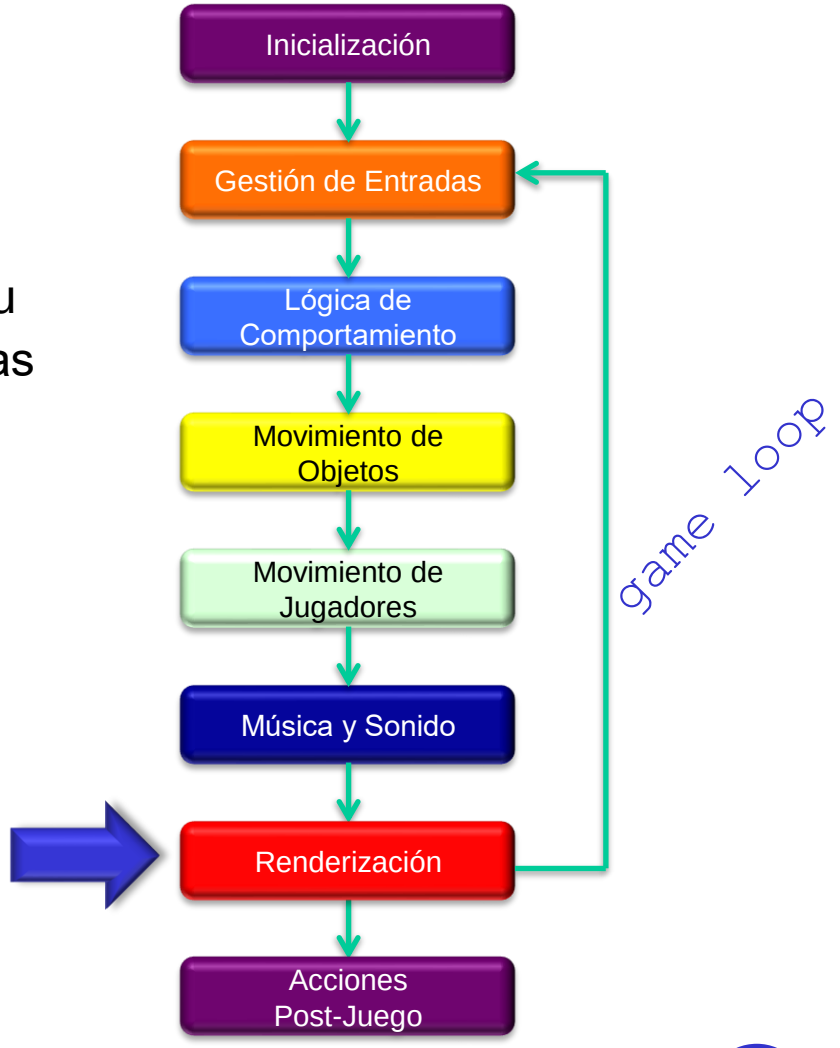


- Música y Sonido:
  - Actualizar variables de efectos de sonido
  - Manejar las variables y acciones de música de fondo

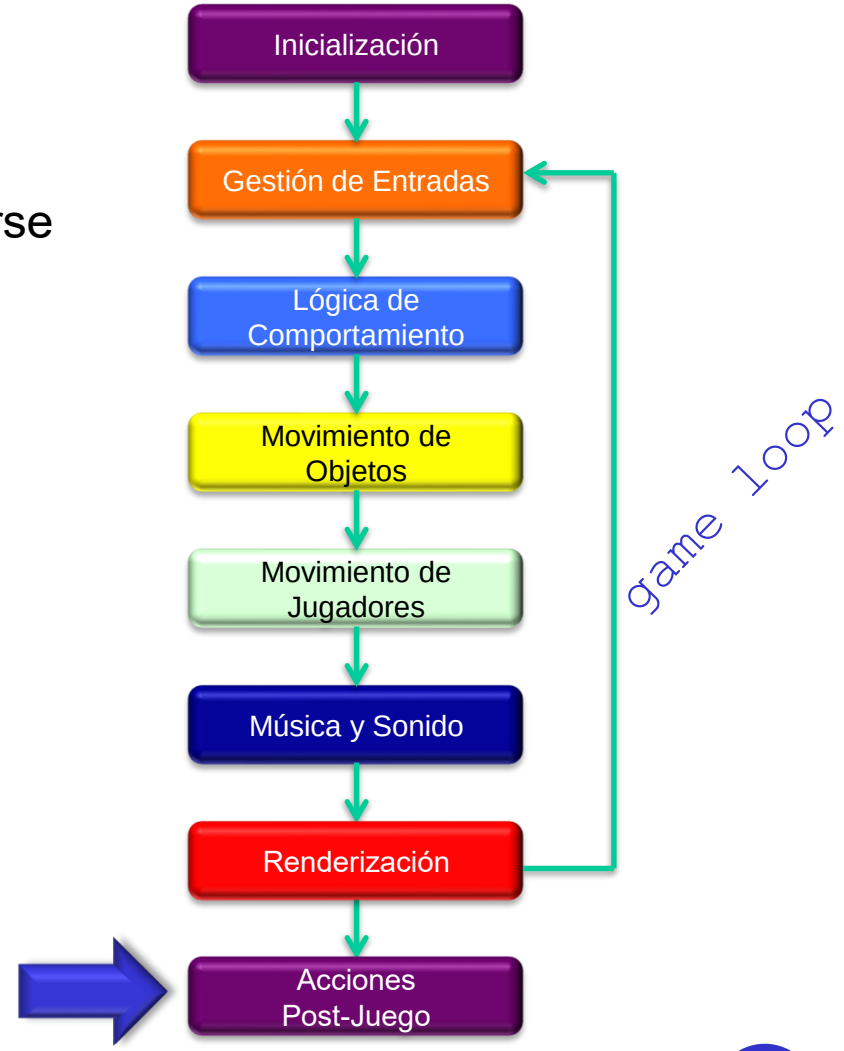




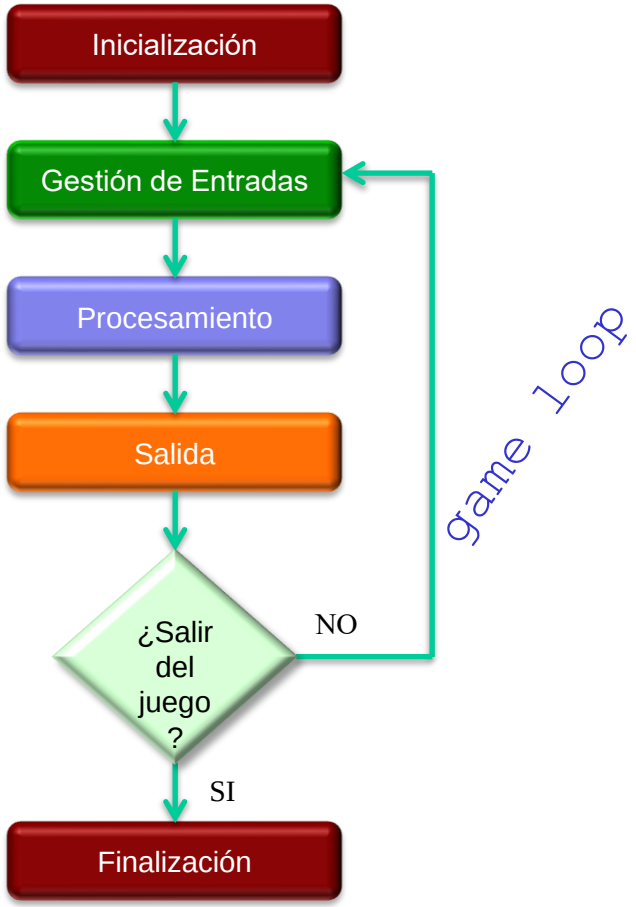
- Renderización:
  - *Imprimir* en pantalla imágenes empezando por el fondo…
  - Continuar con el resto de capas u objetos… desde el fondo hasta las más superficiales
  - Finalmente, personajes o *sprites* de jugadores



- Acciones Post-Juego:
  - Archivar las variables o datos de estructuras que deban recuperarse en un futuro
  - Archivar partida/nivel, etc.
  - Liberar memoria de estructuras temporales
  - Borrar ficheros temporales, etc.
  - Animación/sonido de salida (si procede)

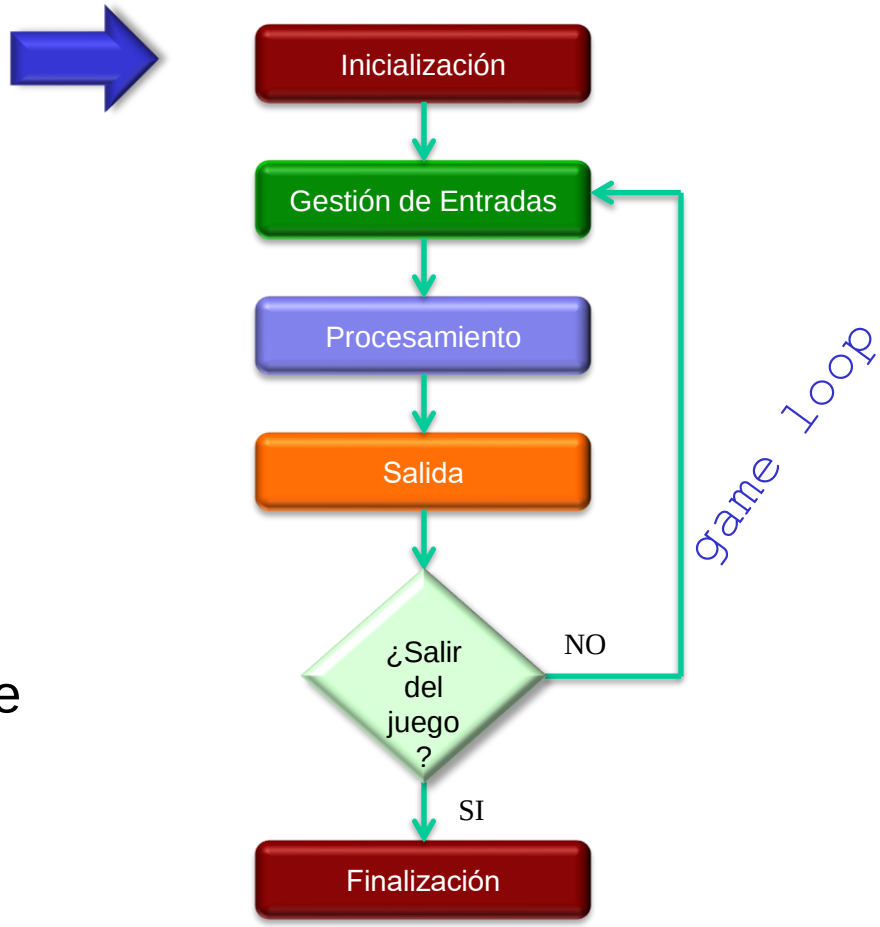


Otra forma de esquematizarlo sería:



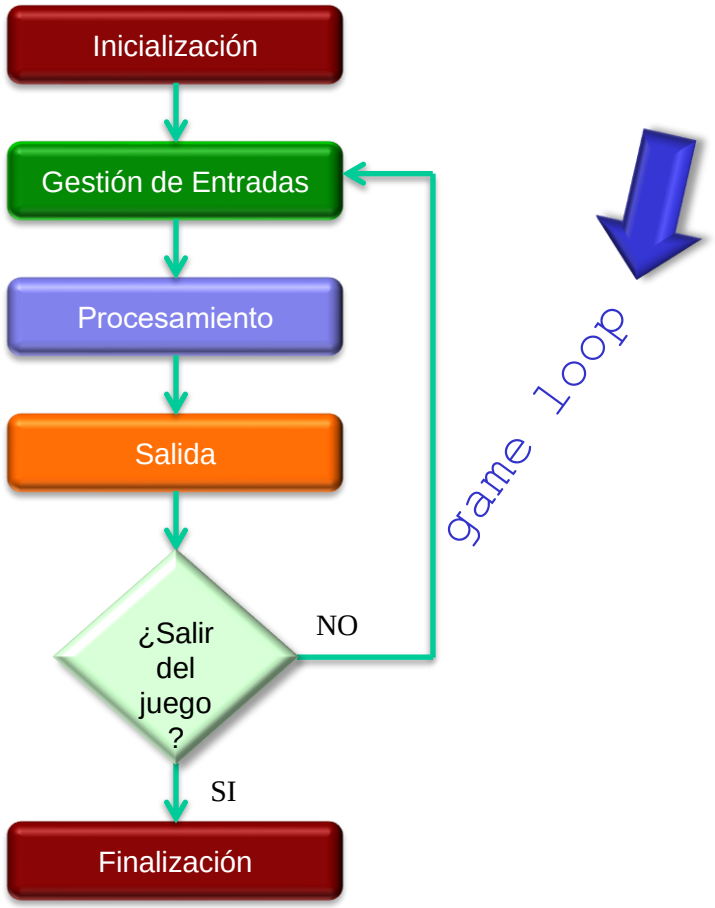
Inicialización:

- Inicialización del subsistema de video
- Estructuras de datos
- Configuración y arranque de bibliotecas
- Inicialización del subsistema de sonido
- Arranque de subsistemas de captura de acciones de usuario
- Posiciones iniciales de usuario



Game loop:

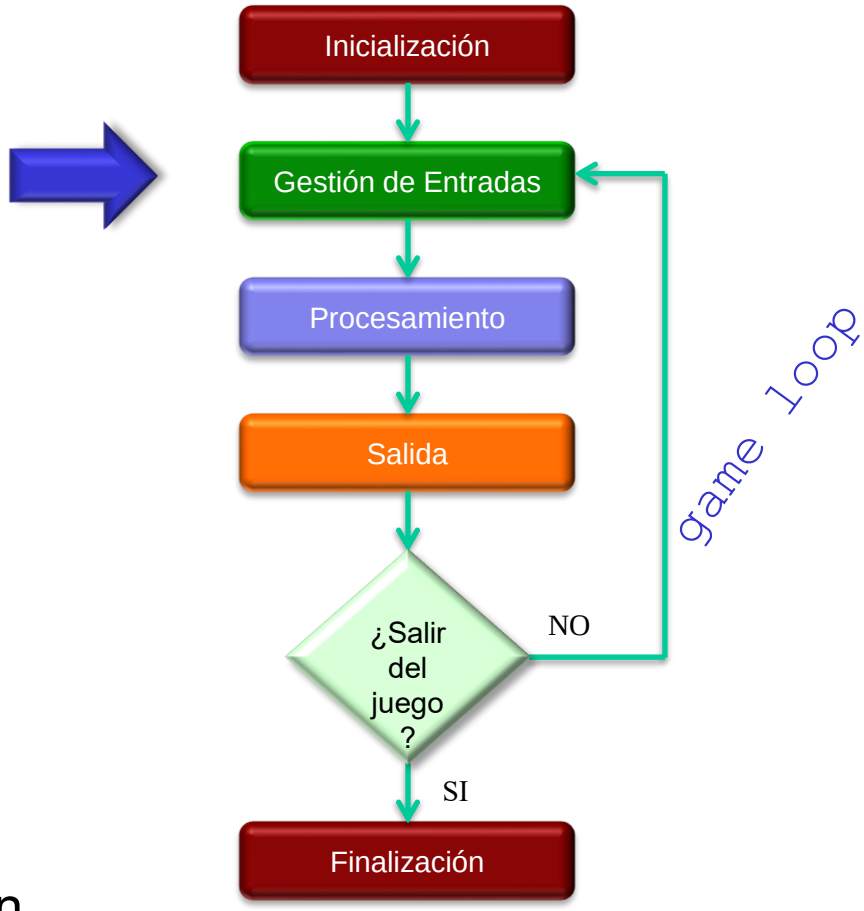
- Gestión de entrada
- Procesamiento
- Salida



Gestión de entrada:

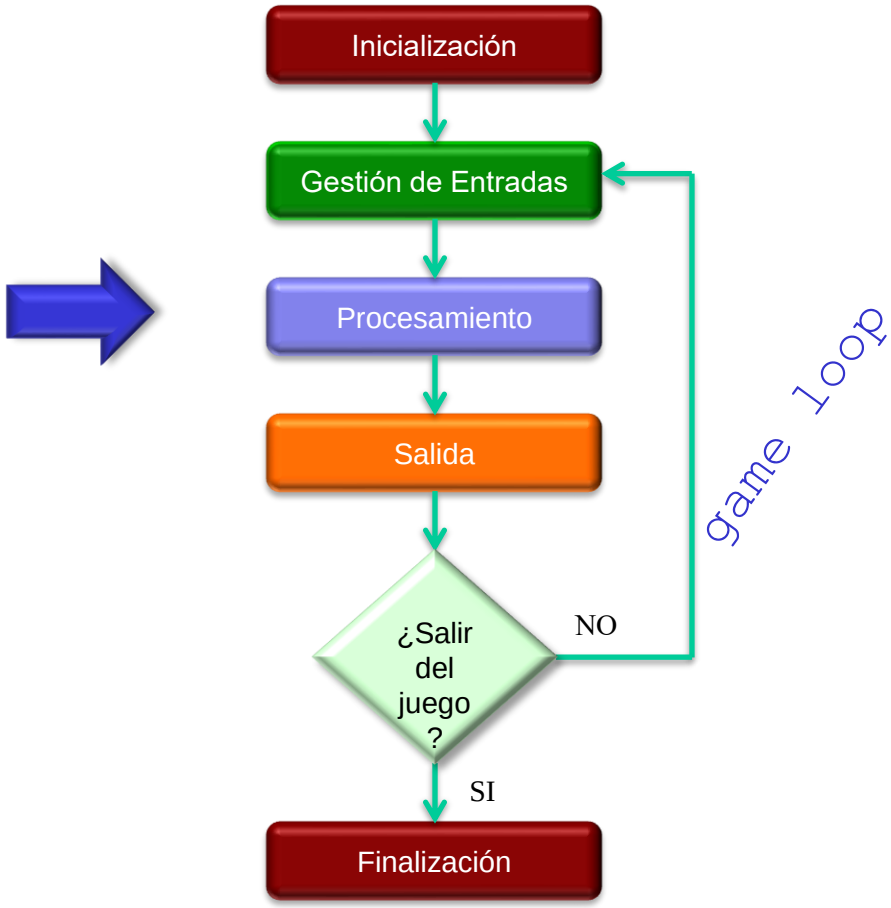
- La gestión de entradas, (o *gestión de eventos*), se obtienen, desde los dispositivos de entrada (teclado, ratón, *gamepad*, etc.), las acciones que ha realizado el usuario

- La *gestión de eventos* es uno de los aspectos fundamentales que determinan el éxito del juego: puede convertirlo en "injugable"



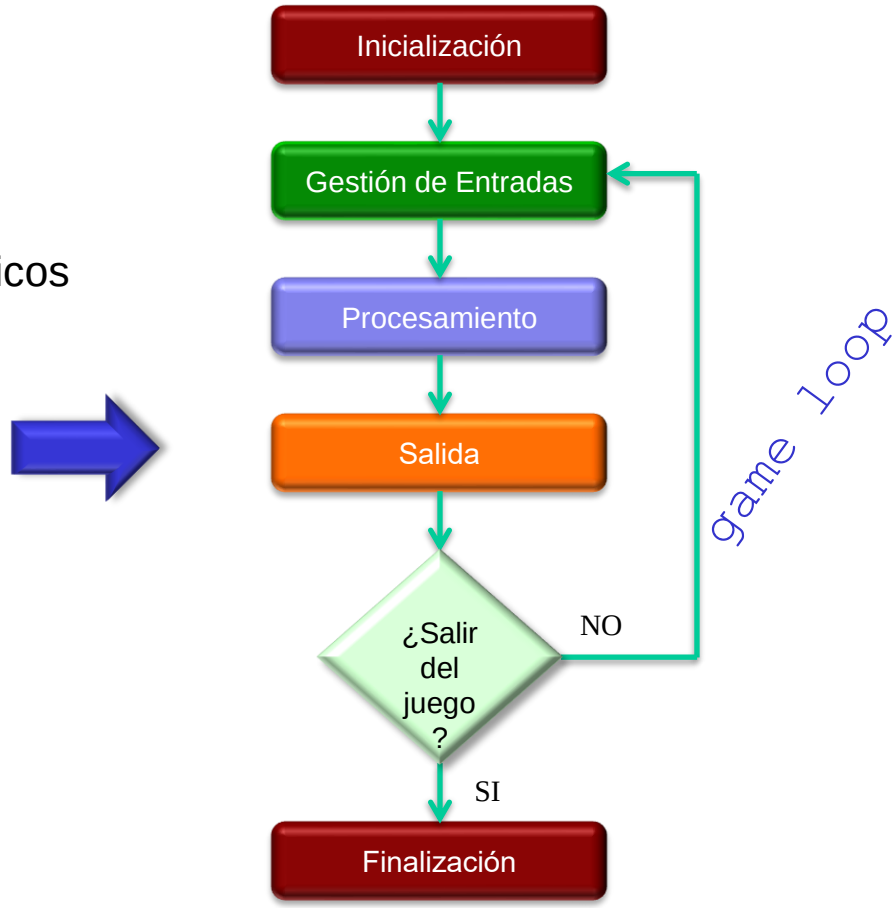
Procesamiento:

- Responde al tipo de entrada que se ha recibido por parte del jugador, tomándose las decisiones adecuadas.
- Actualizar la lógica del juego: Aún si no se ha recibido nada, debe procesarse toda la lógica del juego y los comportamientos que se hayan establecido: IA, gravedad, etc.
- Sincronización de sonido
- Comunicaciones (red)



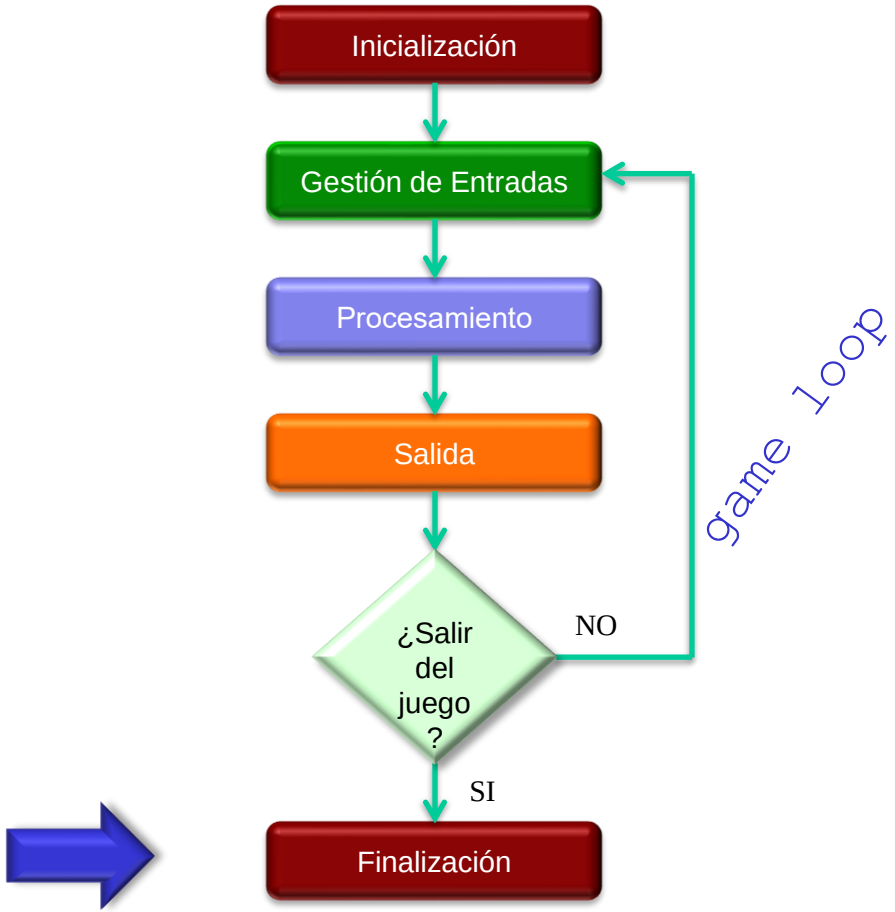
Salida de datos:

- Mostrar al usuario el resultado del procesamiento anterior:
  - Refresco de pantalla: Gráficos
  - Reproducir sonidos



Finalización:

- Tareas opuestas a las realizadas en la Inicialización
- Liberación de recursos



```
#include <iostream>
#include "SDL.h"

using std::cerr;
using std::endl;

int main(int argc, char* args[])
{
    // Initialize the SDL
    if (SDL_Init(SDL_INIT_VIDEO) != 0)
 {
  cerr << "SDL_Init() Failed: " << SDL_GetError() << endl;
  exit(1);
 }

 // Set the video mode
 SDL_Surface* display;
 display = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_DOUBLEBUF);
 if (display == NULL)
 {
  cerr << "SDL_SetVideoMode() Failed: " << SDL_GetError() << endl;
  exit(1);
 }

 // Set the title bar
 SDL_WM_SetCaption("SDL Tutorial", "SDL Tutorial");
```

```
 // Main loop
 SDL_Event event;
 while(1)
 {
  // Check for messages
  if (SDL_PollEvent(&event))
  {
   // Check for the quit message
   if (event.type == SDL_QUIT)
   {
    // Quit the program
    break;
   }
  }
  // Game loop will go here...
 }

    // Tell the SDL to clean up and shut down
    SDL_Quit();

    return 0;
}
```

Ejemplos:

Inicialización del subsistema de video y audio a la vez, y posteriormente establece el modo de video:

```
// Ejemplo de inicialización de subsistemas
// Inicializamos video y audio
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
     fprintf(stderr, "Error en SDL_Init(): %s", SDL_GetError());
     exit(1);
 }
atexit(SDL_Quit); // Al salir, cierra SDL // Modo “chusco”
// Establecemos el modo de pantalla
pantalla = SDL_SetVideoMode(640, 480, 0, SDL_ANYFORMAT);
if(pantalla == NULL) {
// Si no hemos podido inicializar la superficie
     fprintf(stderr, "Error al crear la superficie: %s\n",
     SDL_GetError()); // Captura el último error interno de SDL
     exit(1);
}
```

Ejemplos:

Captura y gestión de eventos de un joystick (1): Inicialización

```
// Si hay un joystick conectado lo abrimos
SDL_Joystick *joy;
if(SDL_NumJoysticks() > 0) {
     joy = SDL_JoystickOpen(0);
     cout << "\nAbrimos el joystick " << SDL_JoystickName(0)
                << " para la prueba. " << endl;
} else {
     cout << "Para llevar acabo esta prueba debe haber un joystick ”
     << "conectado. Si es así compruebe su configuración" << endl;
     exit(1);
}
// Mostramos información del dispositivo
int num_ejes = SDL_JoystickNumAxes(joy);
int num_botones = SDL_JoystickNumButtons(joy);

cout << "Este joystick tiene " << num_ejes << " ejes y ”
                << num_botones << " botones." << endl;

cout << "\nPulse ESC para salir.\n" << endl;
```

Ejemplos:

Captura y gestión de eventos de un joystick (2):

```
// Actualizamos el estado del joystick
SDL_JoystickUpdate();
// Recorremos todos los ejes en búsqueda de cambios de estado
for(int i = 0; i < num_ejes; i++) {
     int valor_eje = SDL_JoystickGetAxis(joy, i);
      //cout << "Eje " << i << " -> " << valor_eje << endl;
      if(valor_eje != 0) {
                i f(i == 0) {
                          if(valor_eje > 0)
                                    posicion.x++;
                          if(valor_eje < 0)
                                    posicion.x--;
                } else {
                          if(valor_eje > 0)
                                    posicion.y++;
                          if(valor_eje < 0)
                                    posicion.y--;
                }
      }
}
```

Ejemplos:

Captura y gestión de eventos de un joystick (3):

```
// Recorremos todos los botones en búsqueda acciones
for(int i = 0; i < num_botones; i++) {
     int pulsado = SDL_JoystickGetButton(joy, i);
                if(pulsado) {
                          cout << "Ha pulsado el botón " << i << endl;
                }
}
```

Cierre del joystick:

```
SDL_JoystickClose(joy);
```

Ejemplos:

Función de finalización del SDL

(finaliza los subsistemas de SDL: video, audio)

El equivalente de SDL_Init es: `SDL_Quit(void);`

```
…
// Finalización más correcta de SDL (si no se usó atexit)
SDL_Quit();
return 0;
}
```

Aspecto general de un *game loop*:

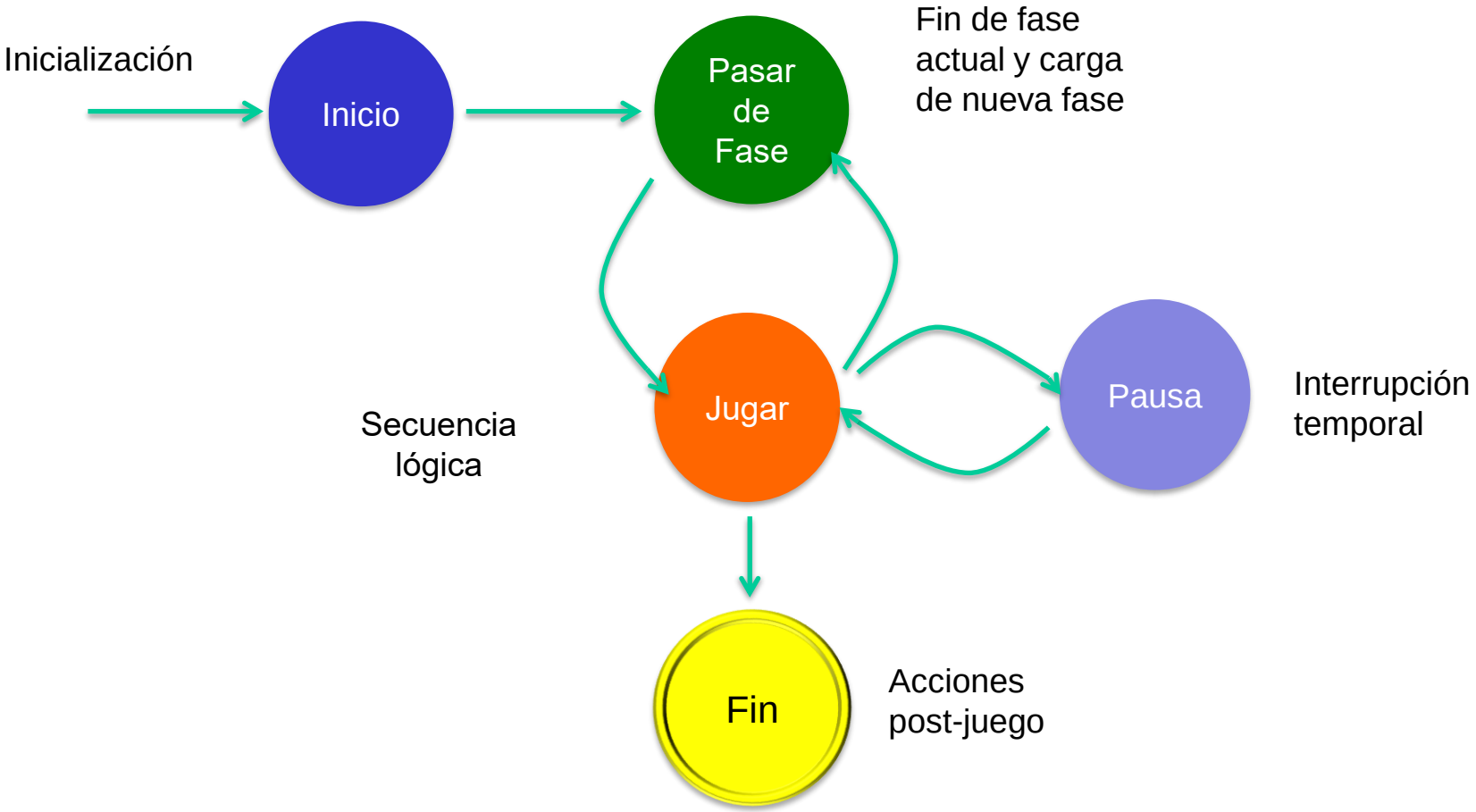
```
int done = 0; while (!done) {
// Leer entrada de usuario
// Procesar entrada (si se pulsó ESC, done = 1)
// Lógica de juego
// Otras tareas
// Mostrar frame
}
```

Índice:

4. Secuencia de Estados

- Autómata Finito Determinístico:
  - Una partida sigue una secuencia de estados intermedios; una evolución del juego.
  - Además, lógicamente puede estar en ejecución, en pausa, cambiando de nivel, iniciando o terminando un nivel, … todo ello se gestiona mediante un autómata finito determinístico.
  - Debe realizarse pues el diseño de dicho autómata definiendo completamente los estados y transiciones entre ellos.

- Autómata Finito Determinístico



Secuencia General (con autómata de estados)

Pseudo-implementación del autómata global de un juego:

```
void Juego(){
        InicializarFase();
        estado=JUGAR;

        mientras (estado != FIN)
                switch(estado) {
                        case JUGAR:
                                EstructuraLogica();
                                break;
                        case PAUSA:
                                esperar();
                                break;
                        case TRANSICION:
                                CambiarFase();
                                break;
                }
        }
}
```

Índice:

5. Introducción a las Herramientas para el Desarrollo de Videojuegos

- Actualmente existen gran cantidad de herramientas para el desarrollo de videojuegos, que abarcan desde:
  - Entornos de programación de propósito general, tales como C/C++, Java, etc., asistidos por bibliotecas (~~librerías~~) para manejo de gráficos (y multimedia) como OpenGL, DirectX, … algunas multiplataforma como SDL
  - Motores de videojuegos (tipo Unreal Engine, Unity, etc.)
  - Lenguajes de script
  - Kits de desarrollo para hardware específico (por ej: videoconsolas)

- Bibliotecas de desarrollo multiplataforma:
  - Ej: SDL (*Simple DirectMedia Layer*)
    - SDL es una API para el desarrollo de juegos, demos y aplicaciones multimedia en general
    - Se puede usar con C/C++ / PHP, etc.
    - Permite programar juegos portables entre plataformas
    - Orientación a objetos
    - Tienen distintos subsistemas asociados para el manejo de video, audio, eventos, hilos-hebras (*threads*), temporizadores, extensiones (archivos de imagen, sonido, acceso a red, fuentes, etc.) mandos de juegos, lector CD/DVD-ROM, etc.
  - Ej: DirectX
    - Es una API para tareas multimedia y programación de juegos, demos, etc.
    - Plataforma Microsoft Windows (en proyecto código abierto para otras)
  - Ej: OpenGL
    - Es una especificación que define una API multilenguaje/multiplataforma
    - Orientada a aplicaciones con gráficos en 2D/3D en general

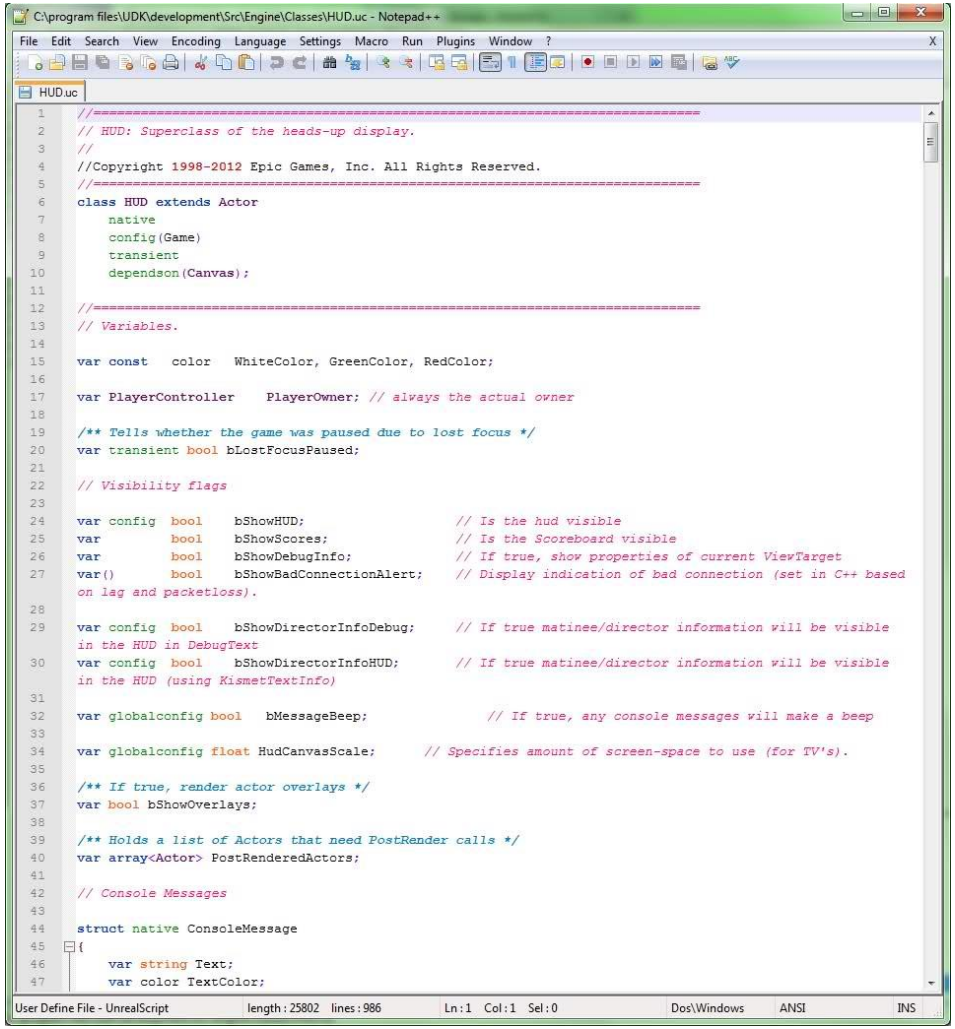
- Motores de Videojuegos (*Engines*):
  - Proporcionan un entorno que simplifica el desarrollo
  - Proveen las diferentes tecnologías necesarias para crear el juego
  - Permiten frecuentemente que el juego se oriente a más de una plataforma con sencillez
  - Incluyen gran cantidad de recursos
  - Historia:
    - Nacieron en 1990 de la mano de los *shooters* en primera persona (*Doom* y *Quake*)
    - Las empresas los adquirían y adaptaban
    - Precios elevados en sus inicios
  - Las grandes empresas crean y evolucionan sus propios motores
  - Hay empresas que son sólo “usuarios” de éstos

- Recursos que incluyen los Motores (I/II):
  - Renderizado para gráficos 2D y/o 3D (partiendo de modelos, generan mediante cálculos la imagen iluminada, sombreada, etc).
    - Con frecuencia, el núcleo de la herramienta son los gráficos, por lo que se les suele llamar genéricamente también *Motores Gráficos*
  - Detección de colisiones
  - Motor de física
  - Sonido
  - *Scripting*
  - Animación
  - Inteligencia Artificial
  - Comunicaciones
  - *Scene graph* (estructuras de datos para gestionar la representación de las escenas)

- Recursos que incluyen los Motores (II/II):
  - Generalmente, el desarrollo parte del protagonismo de los elementos gráficos, es decir, se asocian a él los demás elementos.
  - Algunos muy sencillos, incluso no precisan conocimientos de programación (no hay que escribir código) (aunque es posible en muchos casos hacerlo también) para utilizarlos: típicamente por ejemplo para juegos sencillos de plataforma en 2D, y se pueden usar para diseñar maquetas rápidamente sobre determinadas ideas novedosas previo al desarrollo real.
    - Ejemplos: Stencyl, RPG Maker, Construct 2, Game Maker Studio,
  - Las políticas de venta y precios han cambiado mucho recientemente
    - Libres, de pago, e infinidad de fórmulas mixtas en función de nivel de ventas, versión (prestaciones) del motor, etc.

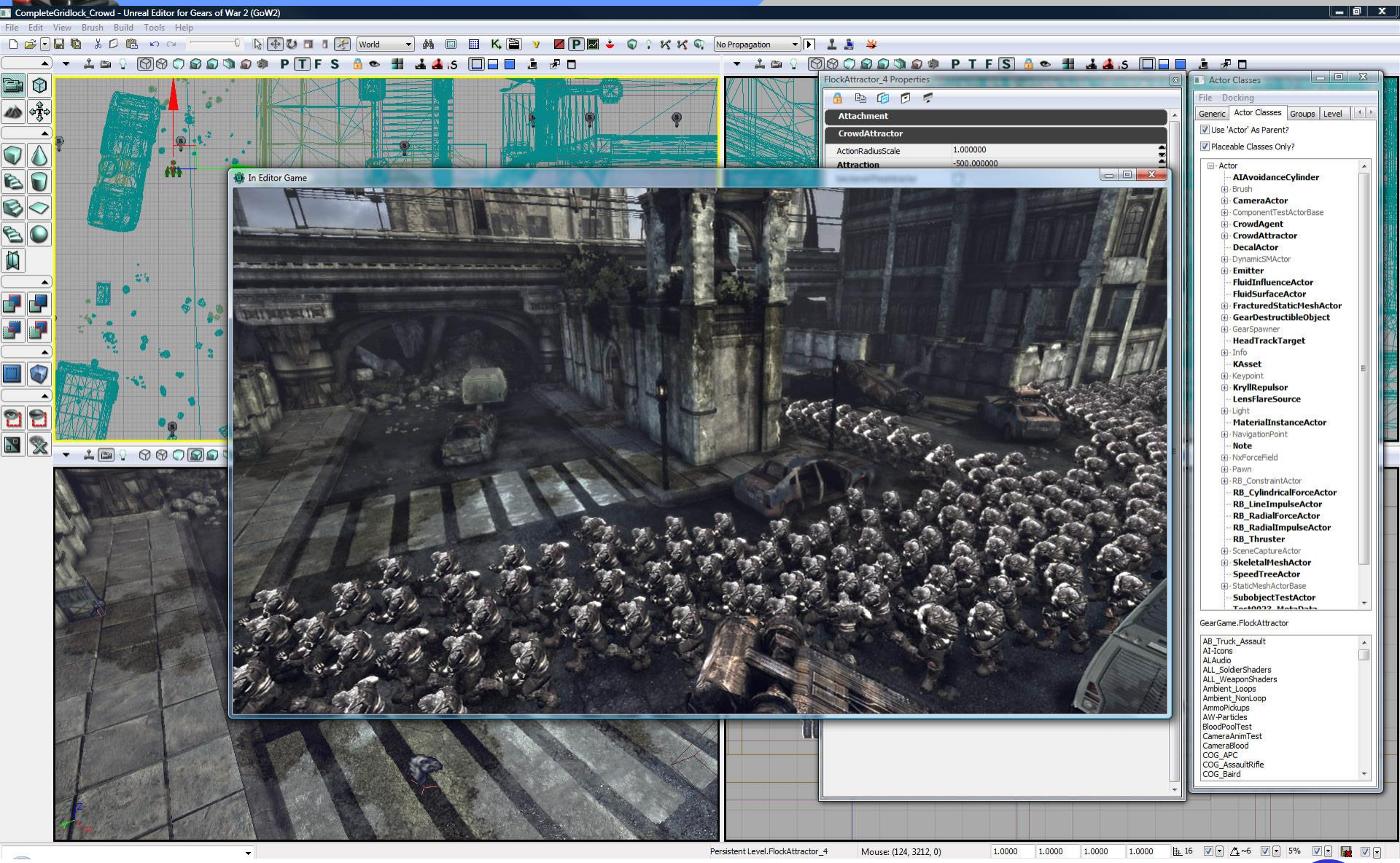
- Ejemplo: Unreal Engine
  - 1998, por Epic Games
  - Mercado profesional: Se ha usado en *shooters* en primera (original) (y tercera persona), para juegos como: *Unreal tournament, Deux Ex, Turok, Gears of War, Star Wars Republic Commando, Mass Effect*, etc.
  - Está escrito en C++
  - Plataformas: Consolas (Dreamcast, Gamecube, Wii, Wii U, Switch, XBOX. XBOX360, XBOX One, PS2, PS3 y PS4, PSVita), PC (Windows/Linux) y Mac OS y OSX, iOS, Android…
  - Utiliza las tecnologías:
    - DirectX 11 y 12
    - OpenGL
  - Actualmente va por la versión 4.xx (orientado a PS4, XBOX One, y PCs “fuertes”)
  - Se ha hecho gratuito, y cobra porcentaje a partir de cierto nivel de ventas
  - Se usa empleando el lenguaje “UnrealScript”

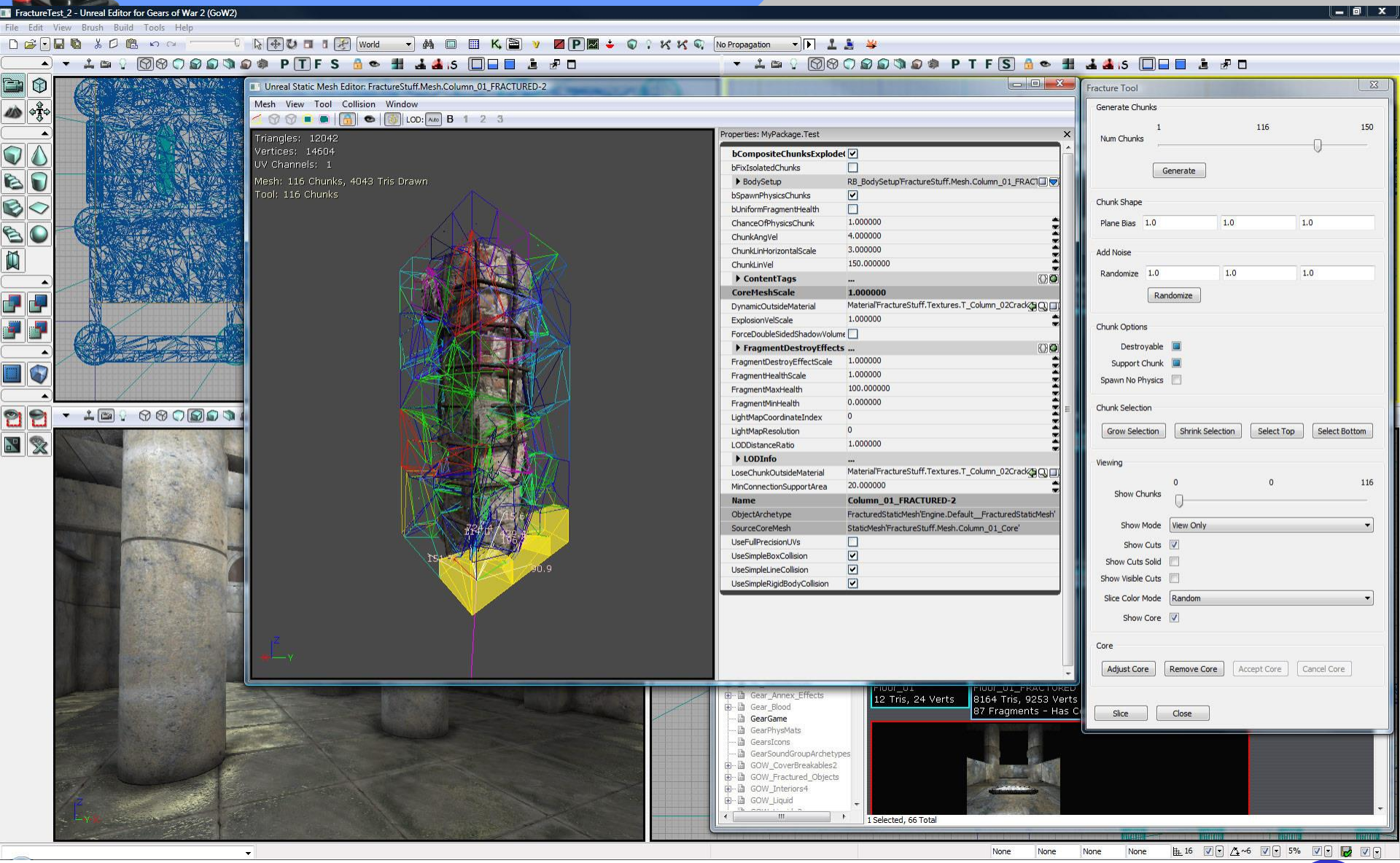
- Ejemplo: Unreal Engine
  - UnrealScript:
    - compilado,
    - orientado a objetos,
    - tipo Java, y
    - con coerción fuerte de tipos en compilación



```
class DMRosterBeatTeam extends xDMRoster;

/*
 * Roster for deathmatch levels.
 * Each level has its own roster.
 * This special roster subclass is used to populate an enemy team with the
 * player's selected teammates.
 */

// called immediately after spawning the roster class
function Initialize(int TeamBots)
{
      local GameProfile GP;
      local int i, j;

      GP = Level.Game.CurrentGameProfile;
      if ( GP == none ) {
                Log("DMRosterBeatTeam::Initialized() failed.  GameProfile == none.");
                return;
      }

      // create roster entries for single player's teammates
      for ( i=0; i<GP.PlayerTeam.Length; i++ )
      {
```

```
                j = Roster.Length;
                Roster.Length = Roster.Length + 1;
                Roster[j] =
class'xRosterEntry'.Static.CreateRosterEntryCharacter(GP.PlayerTeam[i]);
        }

        // remaining team-specific info, might be used in menus at some point
        TeamName = GP.TeamName;
        TeamSymbolName = GP.TeamSymbolName;

        super.Initialize(TeamBots);
}

defaultproperties
{
        TeamName="Death Match"
        TeamSymbolName="TeamSymbols_UT2003.Sym01"
}
```


CompleteGridlock_Crowd - Unreal Editor for Gears of War 2 (GoW2)
File Edit View Brush Build Tools Help
In Editor Game
FlockAttractor_4 Properties
Attachment
CrowdAttractor
ActionRadiusScale 1.000000
Attraction -500.000000
Actor Classes
File Docking
Generic Actor Classes Groups Level
Use 'Actor' As Parent?
Placeable Classes Only?
Actor
AIAvoidanceCylinder
Brush
CameraActor
ComponentTestActorBase
CrowdAgent
CrowdAttractor
DecalActor
DynamicSMActor
Emitter
FluidInfluenceActor
FluidSurfaceActor
FracturedStaticMeshActor
GearDestructibleObject
GearSpawner
HeadTrackTarget
Info
KAsset
Keypoint
KryllRepulsor
LensFlareSource
Light
MaterialInstanceActor
NavigationPoint
Note
NxForceField
Pawn
RB_ConstraintActor
RB_CylindricalForceActor
RB_LineImpulseActor
RB_RadialForceActor
RB_RadialImpulseActor
RB_Thruster
SceneCaptureActor
SkeletalMeshActor
SpeedTreeActor
StaticMeshActorBase
SubobjectTestActor
GearGame.FlockAttractor
AB_Truck_Assault
AI-Icons
ALAudio
ALL_SoldierShaders
ALL_WeaponShaders
Ambient_Loops
Ambient_NonLoop
AmmoPickups
AW-Particles
BloodPoolTest
CameraAnimTest
CameraBlood
COG_APC
COG_AssaultRifle
COG_Baird
Persistent Level.FlockAttractor_4
Mouse: (124, 3212, 0)


FractureTest_2 - Unreal Editor for Gears of War 2 (GoW2)
Unreal Static Mesh Editor: FractureStuff.Mesh.Column_01_FRACTURED-2
Triangles: 12042
Vertices: 14604
UV Channels: 1
Mesh: 116 Chunks, 4043 Tris Drawn
Tool: 116 Chunks
Properties: MyPackage.Test
Fracture Tool
Generate Chunks
Num Chunks
Generate
Chunk Shape
Plane Bias
Add Noise
Randomize
Chunk Options
Destroyable
Support Chunk
Spawn No Physics
Chunk Selection
Grow Selection
Shrink Selection
Select Top
Select Bottom
Viewing
Show Chunks
Show Mode
View Only
Show Cuts
Show Cuts Solid
Show Visible Cuts
Slice Color Mode
Random
Show Core
Core
Adjust Core
Remove Core
Accept Core
Cancel Core
Slice
Close
12 Tris, 24 Verts
8164 Tris, 9253 Verts
87 Fragments - Has C
1 Selected, 66 Total

- Actividad presencial entregable:

¿Estáis familiarizados con algún tipo de herramienta de programación de videojuegos?

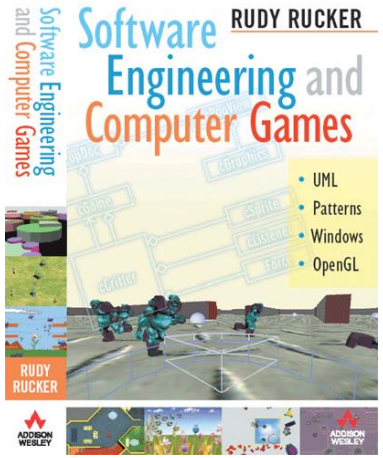
Si es así, redacta tu experiencia.

Si no es el caso, redacta qué esperas de ellas.

6. Bibliografía

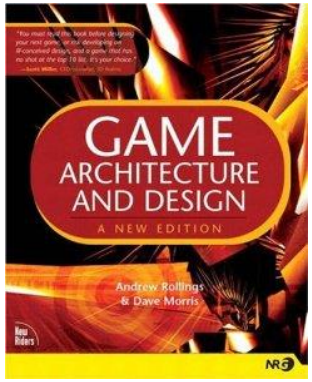
Software engineering and computer games. R. Rucker. Addison-Wesley/Pearson Education 2002

Libre online desde 2011: http://www.rudyrucker.com/computergames/

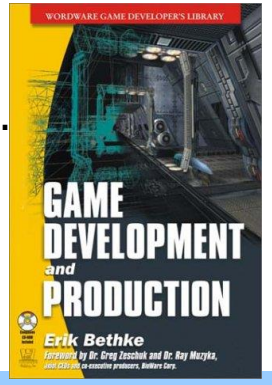


Otras lecturas:

Game Architecture and Design, a New Edition. A. Rollings, D. Morris. New Riders.



Game Development and Production. E. Bethke.



GAME OVER