Modelos y Simulación para Videojuegos II

PROVISORIO

Unidad 4

Simulaciones en Realtime

Ing. Mariano Banquiero

Contenido

Introducción

Una simulación en tiempo real (real time) es aquella en el cual el sistema simula los eventos a la misma velocidad que en la vida real. El tiempo transcurrido en la simulación avanza a la misma velocidad que en la vida real. Por ejemplo si un cierto suceso tarda 1 minuto en completarse en la vida real, también tarda un minuto de simulación. Este tipo de característica hace que las simulaciones en real time sean de suma importancia en el contexto de los videojuegos. En contrapartida los sistemas off-line son aquellos en los cuales el tiempo de procesamiento de las simulación es diferente (usualmente mucho mayor) que el tiempo del proceso físico real. Por ejemplo una simulación del movimiento de un fluido de un minuto de duración puede llevar varias horas o días de cómputo.

Animaciones.

Ya sea que el sistema de simulación sea offline o sea en realtime, eventualmente hay que presentar la simulación por pantalla. Si bien es posible que en una simulación física solo se busquen computar ciertos resultados numéricos, en la mayor parte de los casos nos interesa dibujar una animación real del modelo físico en la pantalla.

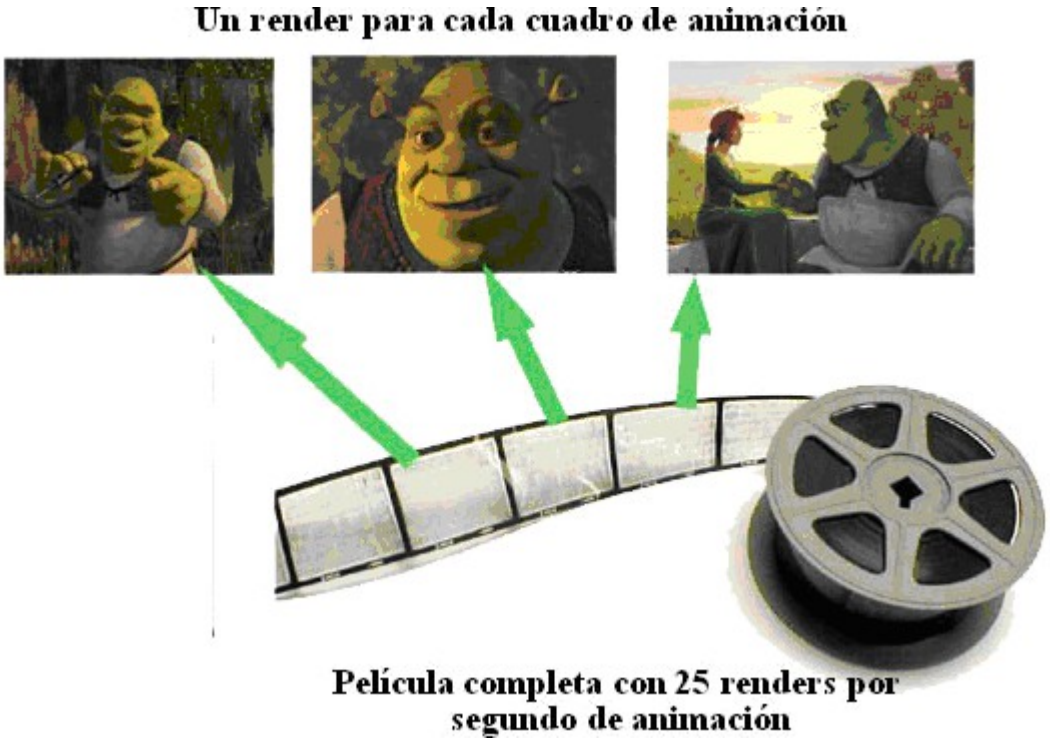
En este caso, es preciso dibujar la escena varias veces por minuto para lograr la sensación de animación. La cantidad de veces que se dibuja la escena se llama fps por sus siglas en inglés frames per second.

La sensación de animación se consigue dibujando la escena varias veces por segundos.

A medida que la cantidad de fps es mayor se logra una mayor sensación de continuidad en los movimientos. A partir de los 30 fps el ojo humano no detecta la transición entre imágenes percibiendo el movimiento como si fuese continuo. Si

bien se considera que 30 fps es un valor aceptable para la mayoría de los videojuegos, con 60 fps se logra la mejor experiencia de jugabilidad.

Las películas de cine ejecutan a 24 fps, sin embargo el método de filmación agrega un efecto llamado motion blur, debido al factor de obturación de la cámara, y eso hace que el ojo humano lo perciba mucho más suave y continuo que el mismo ratio en un sistema de renderizado que no soporte motion blur.



Las películas de cine tienen usualmente 24 o 25 imágenes por cada segundo de animación.

Por lo dicho anteriormente las simulaciones en tiempo real tienen una naturaleza crítica respecto del tiempo, es decir, todos los cálculos de la simulación deben computarse bajo fuertes restricciones de tiempo. Esto hace que el diseño de este tipo de programas este orientado a la performance.

Modelo orientado al control

Cuando comenzamos nuestros primeros pasos en la programación estructurada, se nos enseñó que un programa de computadora debe comenzar y terminar dentro de un bloque de código principal, como por ejemplo la función *main()* en C/C++. Una vez que el programa inicia ejecutando esta función, todo el ciclo de vida del programa ocurre dentro del alcance de este bloque, ejecutando continuamente una instrucción detrás de otra hasta que se alcanza la última instrucción del bloque. Este modelo de programa resulta muy útil cuando el objetivo es realizar un procesamiento con poco interacción por parte del usuario, sin embargo no es óptimo si se requiere que el programa este continuamente

ejecutándose y esperando que ocurran eventos para disparar distintos tipos de procesamiento.

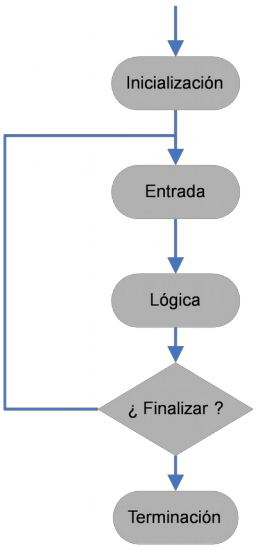
En el modelo orientado al control, cuando el sistema desea obtener los eventos que están ocurriendo en el exterior, este debe consultar los distintos estados de los dispositivos y luego continuar la ejecución del programa. Este muestreo de los dispositivos se realiza continuamente e independientemente si éste estado ha cambiado o no. El hecho de estar consultando continuamente el estado de los dispositivos provoca una alta utilización de recursos del procesador.

Es por eso que en la actualidad los sistemas operativos poseen la capacidad de ejecutar varias aplicaciones simultáneamente, logrando alternar el uso de procesador para cada proceso. Cuando una aplicación está esperando un evento, el proceso queda en estado de espera y el sistema operativo utiliza los recursos de procesador para ejecutar otros procesos. Ya que el modelo orientado al control no cuenta con esa capacidad, este no es muy utilizado en las aplicaciones RTIS ya que el bucle principal monopoliza todos los recursos del sistema evitando que las otras aplicaciones puedan correr apropiadamente. Para remediar esto, necesitamos tener un modelo en el cual se haga utilización del procesador cuando realmente se tenga la necesidad. Es por eso que surge el modelo manejado por eventos.

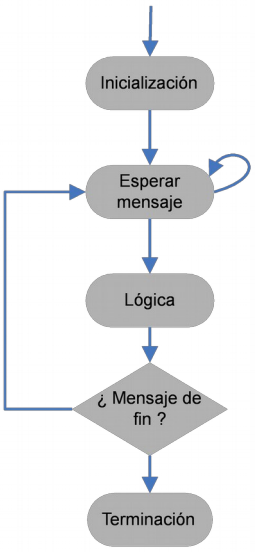
Modelo orientado a eventos

El modelo orientado a eventos soluciona el problema de la eficiencia utilizando una **cola de eventos** o **cola de mensajes**, y reemplazando la función main() por su equivalente MainEventLoop(), pero a diferencia del primero, el bucle de eventos se basa en especificar aquellos eventos que pueden provocar un cambio de estado en la aplicación. Es decir que se definen de antemano los distintos eventos disparadores que deben ocurrir para que el sistema reaccione, realice algún tipo de procesamiento y vuelva a su estado de reposo en espera a próximo evento. Un evento puede ser por el click, o cambio de posición del mouse, una tecla presionada, un temporizador o una interrupción de hardware.

Internamente el subsistema gráfico del sistema operativo mantiene una cola de eventos, donde se encolan todos los eventos de forma cronológica. Es responsabilidad de la aplicación de revisar y procesar ésta cola de eventos en el orden que sucedieron y de realizar un procesamiento en consecuencia. Una vez que esta cola está vacía, y no se quiere realizar acción alguna, el proceso permanece en modo espera, dejando al sistema utilizar los recursos para otras aplicaciones que se encuentren ejecutando.



Modelo orientado al control



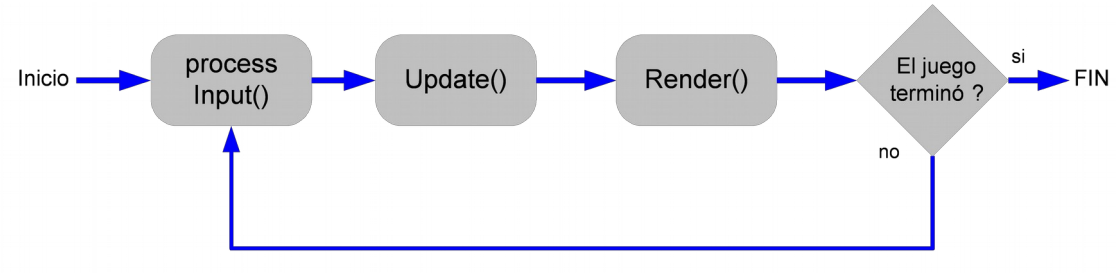
Modelo orientado a eventos

Game loop.

Los videos juegos son aplicaciones orientadas a eventos. Luego de inicializar la aplicación (por ejemplo cargar la escena, las animaciones, mesh de los personajes, etc), el videojuego entra en un bucle continuo donde se procesa la entrada (eventos del mouse, teclado, etc), se ejecuta la lógica de juego (eso incluye la simulación física, la inteligencia artificial, etc) y se dibuja a pantalla. Todo eso se repite continuamente mientras no se de alguna condición de corte (como que el juego termine)

```
while (true)
{
  processInput();
  update();
  render();
}
```

La función processInput() procesa la entrada del usuario, luego la función update() avanza un paso la simulación (esto incluye la inteligencia artificial y la física) y por último la función render() se ocupa de dibujar todo a pantalla para que el jugador vea lo que esta sucediendo.



Bucle acoplado. La velocidad del ciclo depende del hardware.

Es importante tener en cuenta que processInput() procesa la entrada del usuario pero no espera a que se produzca. Por ejemplo, puede preguntar si el usuario presiona cierta tecla y en dicho caso mover el personaje hacia adelante, pero no se queda esperando a que eso suceda. Si en el momento preciso de ejecutar processInput() no hay ningún evento de teclado, joystick, etc., simplemente la función retorna y sigue con el resto de bucle. A este tipo de funciones se las llama no bloqueantes, pues no se queda esperando que se produzca el evento.

```
void processInput()
{
		if(keypressed("W"))
			moveFoward();
		else
		if(keypressed("S"))
			moveBack();
}
```

Entonces como el bucle no es bloqueante las funciones update y render se ejecutarán tantas veces como sea posible dependiendo de la velocidad del procesador. En cada ciclo la rutina update avanzará un paso en la simulación. La cantidad de veces que se ejecuta el ciclo por segundo se llama FPS o frames per second. Si el game loop es rápido y los FPS son altos, el juego se mueve suavemente mejorando la experiencia del jugador.

En este tipo de ciclos en los cuales no hay una sincronización y las funciones render y update se ejecutan tan rápido como es posible, dos factores determinan la cantidad de FPS:

- La cantidad de cómputos a realizar en cada frame especifico, y eso puede variar de frame a frame. Por ejemplo la cantidad de objetos que intervienen en ese momento, la presencia de efectos especiales como explosiones o efectos de partículas costosas para graficar, etc.
- La velocidad del hardware específico donde esta corriendo el juego. Eso incluye la CPU, la GPU, memoria, etc.

En los primeros videojuegos la velocidad del hardware estaba mas o menos fija. Si se escribía un juego para una commodore 64 o una NES, se sabía exactamente a que velocidad corría la CPU y el código se adaptaba o "tuneaba" a

esa configuración específica. El juego estaba diseñado para ejecutar una cierta cantidad de cómputos de tal forma que la velocidad de ejecución sea la correcta.

Commodore 64.

Por ejemplo, el siguiente código pregunta si está presionada la tecla 'S', y en ese caso le suma 10 (pixeles) a la posición x del personaje. De manera similar si esta presionada la tecla 'A' le resta 10 unidades. Es decir esta usando las teclas S y A para mover al personaje a derecha e izquierda por la pantalla.

```
function update()
{
   if(key=='S') pos_x = pos_x + 10;
   if(key=='A') pos_x = pos_x - 10;
}
```

Queda claro que la velocidad del personaje en pantalla dependerá de la cantidad de veces que se ejecute la función update dentro del game loop. El valor de 10 pixels tuvo que ser cuidadosamente determinado para obtener la velocidad deseada. Si este mismo código se ejecutara en un hardware más rápido, la velocidad del jugador sería proporcionalmente más rápida, inclusive podría dificultar la jugabilidad hasta el punto de hacer imposible jugar el juego. Por este motivo los simuladores de home computers de 8 bits para PC usualmente simulan también la velocidad del reloj del procesador, de tal forma que los juegos escritos para dichas computadoras corran a la misma velocidad que en el hardware original.

Botón de turbo:

Antiguamente, en la época de los 8086 y 8088, la forma de programar era bastante diferente que la de ahora. El acceso al hardware era mucho más directo, y, prácticamente, un programador debía ser un experto en hardware.

Uno de los problemas más comunes, y que no muchos programadores tuvieron en cuenta (o bien, no supieron cómo hacerlo con las herramientas que tenían en esa época) era la posibilidad de ejecutar sus programas, diseñados para un procesador de 4 a 8 MHz en un equipo más rápido, corriendo a 16 MHz o más. En aplicaciones con cálculos avanzados, los programadores suponían el tiempo que tardaría el procesador en hacerlos, y, mientras tanto, realizaban otras operaciones. Pero, si el procesador utilizado es mucho más rápido de lo previsto, es probable que requiera de algún dato que aún no está disponible. Si esto ocurre, se generará un error que hará inútil al programa.

Otro caso típico es de los juegos: al no tener un limitador de cuadros por segundo, en procesadores rápidos la velocidad se iba a las nubes, y era casi imposible jugar.

Así que, para estos casos, había que diseñar un método que, de alguna manera, redujera la performance del equipo, a fin de que estas aplicaciones se pudiesen ejecutar. Y esa era la función del Turbo: reducir la velocidad del equipo. Si el botón estaba "encendido", o bien se encontraba desconectado del motherboard, se trabajaba a la velocidad normal (por ejemplo, en nuestro caso del 386, 40 MHz). De lo contrario, si se pone en el estado "no Turbo", se reduce la velocidad "en caliente", es decir, instantáneamente. El método más común de bajar la velocidad, en 286 y 386 es simplemente dividir por dos la frecuencia del procesador (en

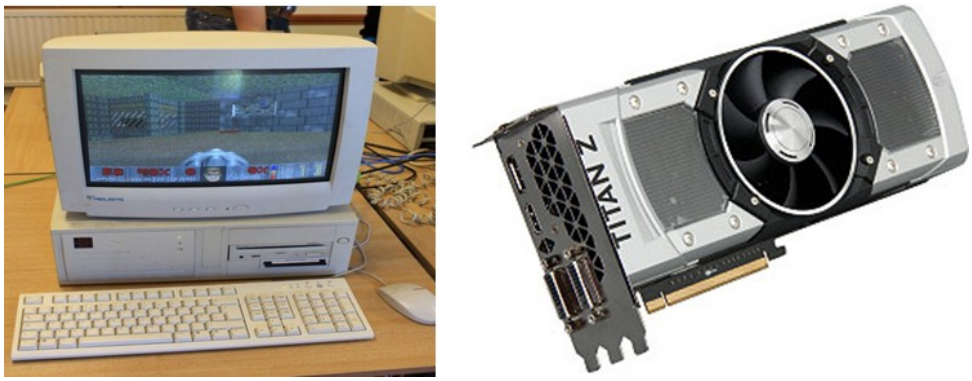
este caso, 20 MHz). En el caso de los 486, lo que se solía hacer era agregar tiempos de espera, o deshabilitar la caché, para bajar el rendimiento.

El final del Turbo fue justamente en esta generación, siendo totalmente inútil en los Pentium y superiores (ya se dejó de "soportar" al software anticuado). Así que, si ven algún Pentium con botón Turbo, sepan que no es más que para adornar estéticamente.

Hoy en día muy pocos programadores tienen el lujo de conocer con exactitud el hardware específico donde va a ejecutar el videojuego. En estos casos, el programa debe ser diseñado para adaptarse a las distintas plataformas de forma transparente al usuario. Unas de las funciones más importantes del game loop es permitir al juego correr de la misma forma en las distintas arquitecturas, independizando la velocidad de las animaciones de la velocidad del hardware específico.

Bucle desacoplado. Timesteping.

Lo que precisamos es una forma de controlar el tiempo transcurrido entre distintos updates de tal forma que la lógica del juego (IA, simulaciones, etc.) dependa de ese tiempo y no de la velocidad del hardware específico. A esto se lo llama timesteping, y es lo que nos permite programar un videojuego que ejecute a la misma velocidad en todas las plataformas.



Variedad de hardware con muy diferentes velocidades de procesamientos.

Introduciendo la variable tiempo transcurrido llamada "elapsed_time" en el game loop podemos modificar la rutina update para que tenga en cuenta dicho tiempo transcurrido a la hora de resolver las ecuaciones de movimiento y demás ecuaciones físicas que depende de la variable tiempo.

El siguiente código muestra la rutina update para mover al personaje por la pantalla adaptada un ciclo desacoplado del hardware. Al utilizar una variable con el tiempo transcurrido desde la última llamada, el movimiento del personaje se hace multiplicando un cierto valor (en este caso 50 pixeles) por el tiempo transcurrido. De esta forma el personaje siempre se mueve a 50 pixeles por segundos, una velocidad constante que es independiente del hardware específico.

```
function update(float elapsed_time)
{
  if(key=='S') pos_x = pos_x + 50 * elapsed_time;
  if(key=='A') pos_x = pos_x - 50 * elapsed_time;
}
```

En un hardware más rápido la función update se va a ejecutar más veces, sin embargo, con menos elapsed_time. En un hardware más lento se va a ejecutar menos veces pero con mayor elpased_time. El resultado es que el personaje se mueve siempre a la misma velocidad.

Bucle de tiempo Fijo. Fixed time Step.

En el ejemplo anterior, si bien el personaje se movía siempre la misma velocidad, el bucle podía ejecutar una cantidad de fps variable. Por ejemplo, en alguna parte del juego podría estar funcionando a 60 fps y en otra a 200 fps (porque por ejemplo, no había demasiados objetos para renderizar). Eso puede traer alguna sensación de discontinuidad en el juego, estudios recientes demostraron que el jugador prefiere que el juego se renderize siempre a la misma velocidad aun cuando sea más lenta, y no que se renderize a la máxima velocidad posible. Es decir que es preferible que ande a 30 fps constantes, a que funcione a 40 fps en algunos lugares y en otros a 120 fps. Independientemente de esta cuestión que puede llegar a ser subjetiva, es conveniente que el motor de física funcione a una velocidad constante. De esta forma se simplifican mucho las ecuaciones del motor, además se evitan problemas de redondeo y es más fácil debuggear si el elapsed time es fijo en cada update.

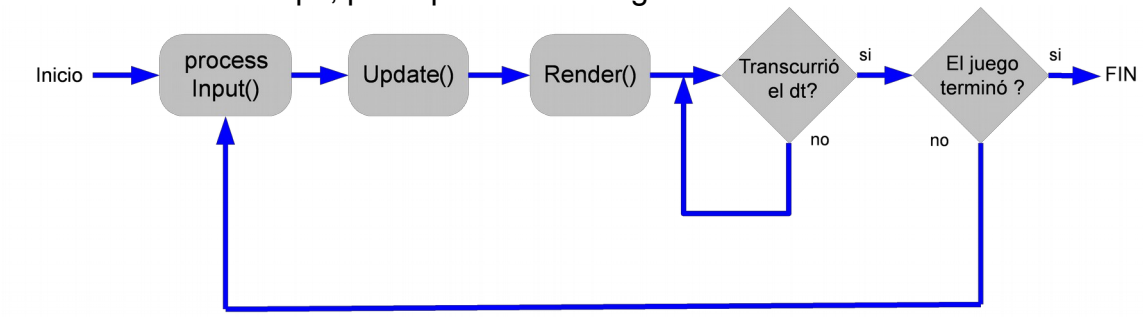
Supongamos que queremos que el juego corra a una velocidad constante de 30 fps, es decir queremos que las funciones render y update se ejecuten unas 30 veces por segundo. Esto quiere decir que se dispone de 32 milisegundos por frame. Mientras sea posible ejecutar todo el cálculo (física, IA, etc) y graficar por pantalla en un lapso de tiempo menor o igual a 32 ms, podemos lograr un ciclo constante de 30 fps agregando una pausa al final que espere sin hacer nada hasta que se complete dicho tiempo. Entonces el game loop hace el procesamiento y el rendering y luego espera sin hacer nada hasta que se complete el tiempo por frame predeterminado y fijo.

```
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();

  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

La función sleep nos asegura que el game loop no ejecutará más rápido que los MS_PER_FRAME que tiene predefinido. Sin embargo, nada se podrá hacer si el juego ejecuta demasiado lento. Si por algún motivo la función render o update no logran completarse en el tiempo preestablecido, la función sleep no esperará nada (ya que recibirá un tiempo de espera negativo!), pero el bucle no ejecutará a la velocidad de fps requerida y el jugador experimentará un lag o una sensación de discontinuidad en las animaciones.

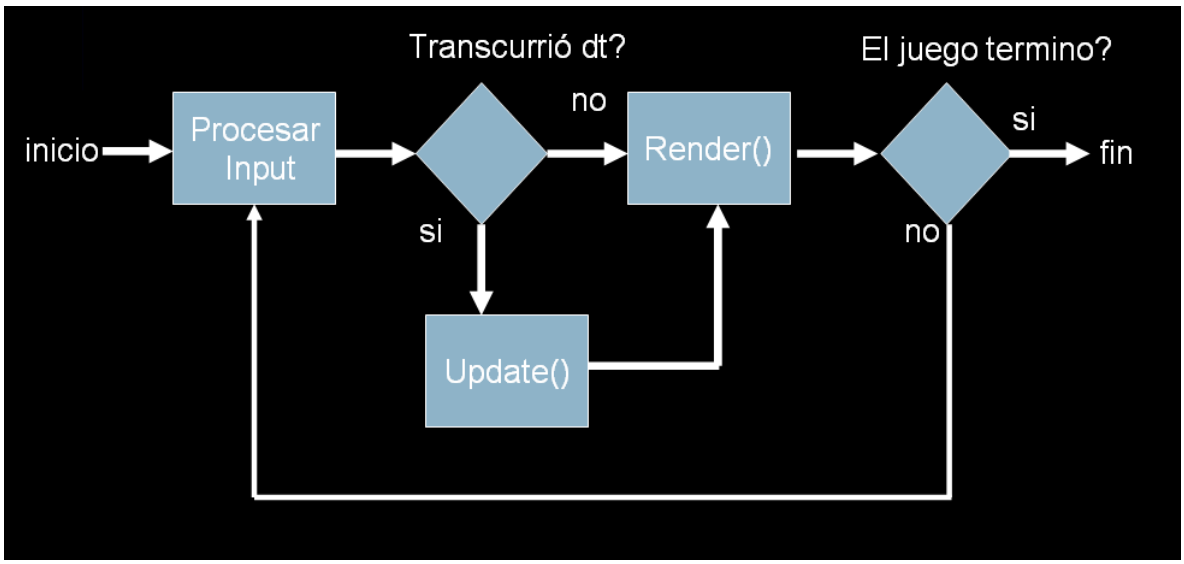
En ese caso tendremos que revisar la lógica del update para simplificarla, optimizarla o reducir la cantidad de trabajo por frame de tal forma que el juego pueda correr a esa velocidad. O bien bajar los fps, por ejemplo si vemos que no es posible ejecutar el game loop a una velocidad constante de 60 fps, quizá sea conveniente correrlo a 30fps, pero que se mantenga constante.



bucle de tiempo fijo. El bucle se ejecuta a una cantidad fija de fps

Update y Render Desacoplado.

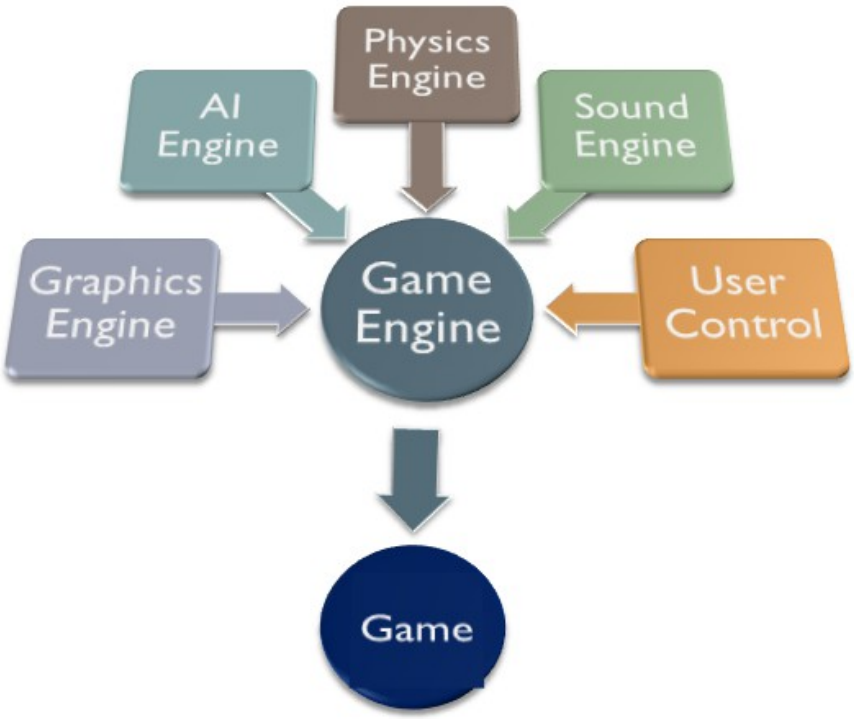
Muchas veces puede ser conveniente desacoplar la física del juego de la función de graficado. La idea es que la física se llame cada cierto elapsed time fijo mientra que la función de graficado se puede llamar tantas veces como se pueda.



Variante de Game loop: función render y update desacopladas.

Game Engine

Un motor de videojuego o game engine es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Del mismo modo existen motores de juegos que operan tanto en consolas de videojuegos como en sistemas operativos. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico. El proceso de desarrollo de un videojuego puede variar notablemente por reutilizar o adaptar un mismo motor de videojuego para crear diferentes juegos.



Componentes básicos de un motor de video juegos.

Hoy en día existen una gran variedad de motores completos y motores gráficos como OGRE 3D que es un motor gráfico gratuito con "open-source" para que el usuario pueda crear aplicaciones desde el lenguaje C++. Desarrolladoras grandes de videojuegos como Epic, Valve y Crytek han lanzado al público sus motores o SDKs para que los usuarios interesados en el desarrollo de videojuegos puedan descubrir como se elaboran y así tener una introducción amplia a la industria y el desarrollo. Otros ejemplos de motor de juego seria el motor gráfico Doom engine, Quake Engine, GoldSrc desarrollado por Valve en el cual se

desarrolló el exitoso juego de Half-Life 1, Source también creado por VALVe y BLAM! Engine desarrollado por Bungie lo cual crearon la famosa saga de Halo.

Algunos motores de video juegos más conocidos.

Graphic Engine.

El motor gráfico, graphic engine o render engine es el encargado del proceso de graficación o renderización, es decir de mostrar en pantalla el aspecto visual de nuestro juego. El render se encarga de mostrar al jugador todo el poder gráfico que el desarrollador haya configurado en el motor, el render muestra todo lo que es el Terreno o BSP, Modelos, animaciones, texturas y materiales. El render contribuye todo el aspecto visual del juego.

Assets

Los assets pueden ser traducidos como elementos que serán introducidos al videojuego. Estos elementos incluyen Modelos 3D, personajes, texturas, materiales, animaciones, scripts, sonidos, y algunos elementos específicos de cada motor. Cada motor trabaja de una manera distinta a otros lo cual puede aceptar "Assets" que otros motores no pueden manejar, sin embargo los ejemplos mencionados antes, son elementos que todos los motores de hoy en día usan.

Scripting Systems o Sistemas de scripting

El scripting le permite al diseñador tomar mando de la escena y manipularla, como colocar objetos o eventos que el jugador no controla. La mayoría de estos scripts se basan en lenguaje C. Algunos motores de videojuegos utilizan interfaces visuales para definir el scripting, los llamados Visual Scripting Systems

Sonido

Para el procesado de sonido es muy similar al procesado de los modelos, muchas veces un software los procesa antes de pasar al hardware respectivo, por ejemplo DirectSound hace al sonido para la Tarjeta de sonido lo que Direct3D hace al modelado antes de llegar al la Tarjeta 3D. Esto es llamado “premezcla” en el software.

Hoy en día los motores de nueva generación soportan muchos formatos de sonido pero los más populares son el ".wav" y ".ogg" y en algunas casos, exigen configuraciones exactas dependiendo el motor. La administración de pistas de audio larga son amplias puesto que motores de nueva generación permiten modificación para poder meter un "looping" a la pista, modificar el tono, etc.

Inteligencia Artificial (IA o AI)

Es la característica más importante que se le atribuye a un motor al lado de la representación de modelos o Render. La IA provee de estímulo al juego. Es crítico en la parte de la forma de juego (game play).

La inteligencia artificial de determinado juego puede tornarse muy compleja, primero se debe definir la línea base del comportamiento de los NPC (Non Player Characters - Personajes no Jugables), primero debe definirse qué hace el NPC (patrulla, guarda, etc.), luego se delimita su “visión del mundo”, que es lo es el NPC puede ver del mundo del juego; se debe tomar en cuenta que el personaje no sólo estará en medio del mundo del juego sino que también interactuará con él, después vienen las rutinas de Toma de Decisión: si el NPC está patrullando, y hay un sonido, ¿debe darle importancia o no?, ¿investiga su origen o no?, etc.

En un concepto más general de IA, es un sistema de reglas para las acciones que responden (o inician) y que el jugador debe responder.

Physics Engine

Es la parte del game engine encargada de hacer las simulaciones físicas, como la dinámica de cuerpos rígidos, simulación de daños, detección de colisiones, efectos de partículas, etc.

Damage simulation.

Algunos motores de física son de uso general y pueden simular cuerpos rígidos, partículas, y permiten conectar cuerpos rígidos entre si para simular objetos mas complicados como un ragdoll. Otros en cambio son mas bien específicos y están orientados a simular con mayor exactitud un sistema físico concreto, como puede ser un simulador de vuelo o el simulador de física de un videojuego de futbol.