10. The Game Loop and Real-time Simulation

Game Engine Basics

Prof. H. Kang

Introduction

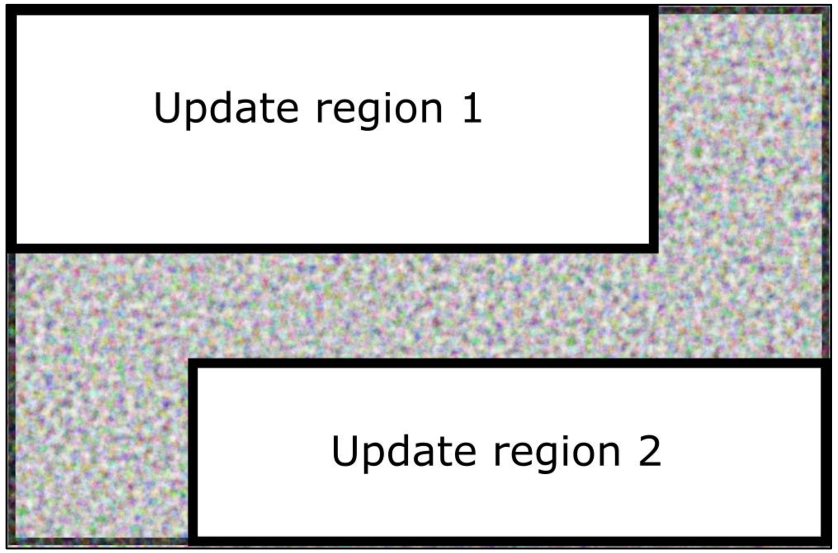
‘Time’ in game

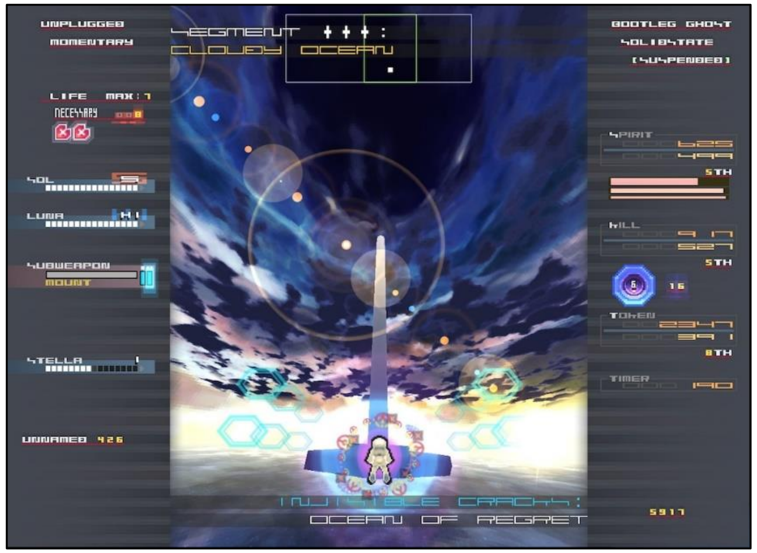
- Since games are real-time, dynamic, interactive computer simulations, *time* plays an important role in game.
- There are many different kinds of time to deal with in a game engine
  - Real-time
  - Game time
  - Local time
  - Animation time
  - ...
- Every engine system define and manipulate *time* differently.
- Therefore, we must have a solid understanding of all the ways *time* can be used in a game.

Rendering Loop

Rendering loop?

- In a graphical user interface (GUI) of the PC, the majority of the screen's contents are static.
  - Only a small part of any one window is actively changing appearance at any given moment.
  - Therefore, GUI have traditionally been drawn on screen via a technique known as *rectangle invalidation*, in which only the small portions of the screen whose contents have actually changed are redrawn.





Rendering Loop

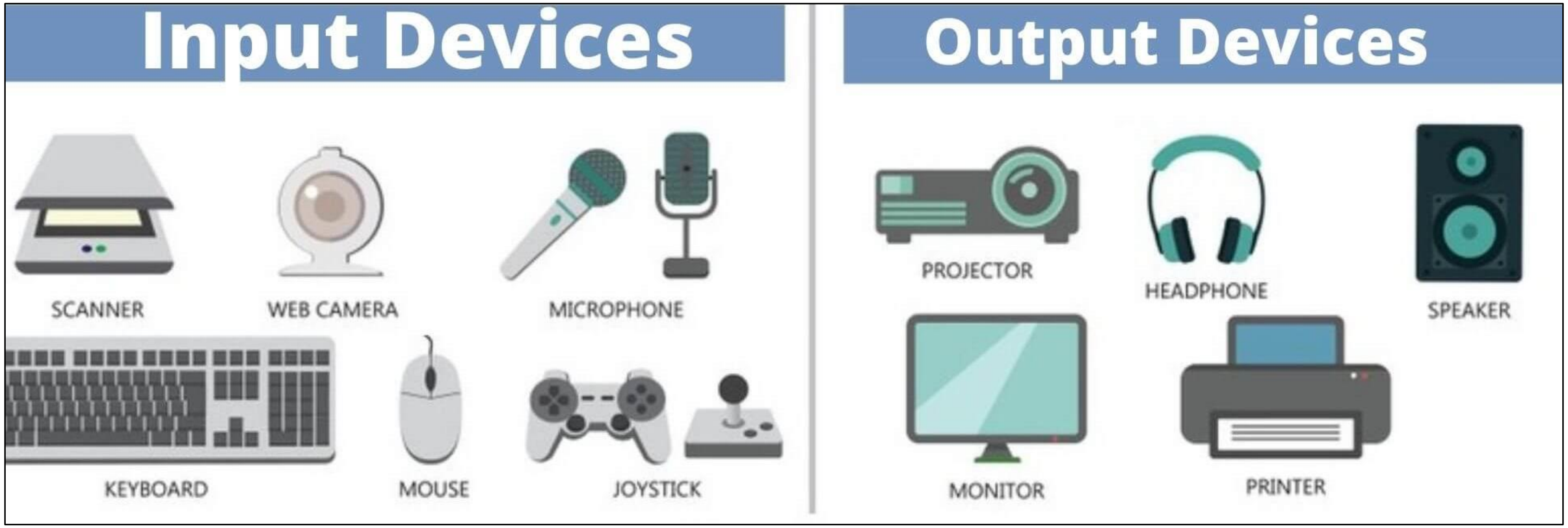
Rendering loop?

- In contrast, the entire contents of the screen change continually in 3D real-time 3D graphics applications.
  - The scene is captured by the camera in the 3D space and the camera moves continually.
  - An illusion of motion and interactivity are produced by presenting the viewer with a series of still images in rapid succession.
  - This requires a *loop* which is known as the *render loop*.

Game Loop

Game loop?

- A game is composed of many interacting subsystems, including device I/O, rendering, animation, collision detection and resolution, optional rigid body dynamics simulation, multiplayer networking, audio, and etc.
- Most game engine subsystems requires periodic servicing while the game is running.



Game Loop

Game loop?

- However, the rate at which these subsystems need to be serviced varies from subsystem to subsystem.
  - Animation typically needs to be updated at a rate of 30 ~ 60 Hz, in synchronization with the rendering subsystem.
  - A dynamics (physics) simulation may actually require more frequent updates (e.g., 120 Hz).
  - AI system might only need to be serviced once per second.
- The simplest way to update the engine's subsystem is to use a single master loop. We often called this *game loop*, and it updates subsystem in each loop.
  - InitGame()
  - GameUpdate()
  - AnimUpdate()
  - inputUpdate()
  - render()

Game Loop Architecture

Game Loop Architectural Styles

- Game loops can be implemented in a number of different ways.
- The core idea is to boil down to one or more simple loops.
- Most game engine subsystems and third-party game middleware packages are structured as libraries.
  - A library is a suite of functions and classes that can be called in any way the application programmer sees fit.
  - It provides flexibility to the programmer, but difficult to use, because the programmer must understand how to properly use the functions and classes.

Game Loop Architecture

Game Loop Architectural Styles

- Some game engines and game middleware packages are structured as frameworks.
  - A framework is a partially constructed application - the programmer completes the application by providing custom implementations of missing functionality within the framework.
  - Programmers has little or no control over the overall flow of control within the application, because it is controlled by the framework.

Game Loop Architecture

Game Loop Architectural Styles

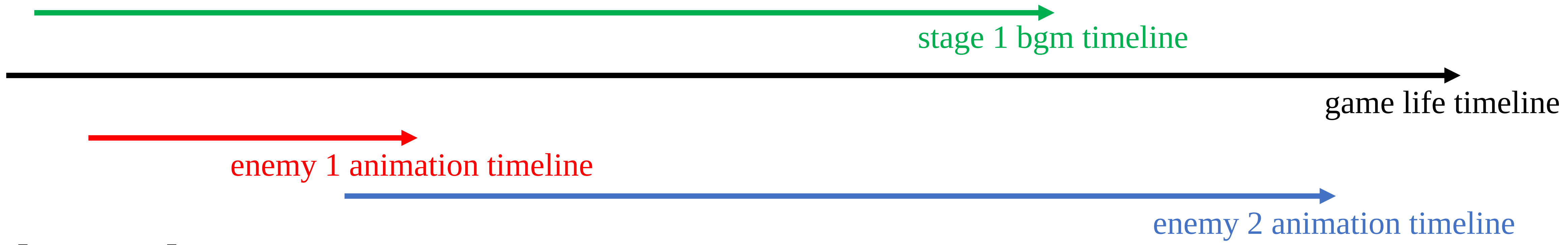
- In a framework-based rendering engine, the game loop has been structured, but it is largely empty.
  - The game programmer can write callback functions in order to fill in the missing details.
  - In this case, programmers should implement *callback* and *callback*_listener. (some engines guided how to implement *update callback* efficiently.

- Event-based updating is also widely used.
  - An event is any interesting change in the state of the game or its environment.
  - Event handling is usually similar to the event/messaging system.
  - For example, to perform a periodic servicing, game engine posts a new event/message every second. Then, event handler responds to the posted event/message.

Timelines

Timeline?

- A timeline is a continuous, one-dimensional axis whose origin ($t=0$) can lie at any arbitrary location relative to other timelines in the system.
- A timeline can be implemented via a simple clock variable that stores absolute time values in either integer or floating-point format.



Real Timeline

- The origin of this timeline is defined to coincide with the moment the CPU was last powered on or reset.
- It measures times in units of CPU cycles (which can be easily converted into units of seconds).

Timelines

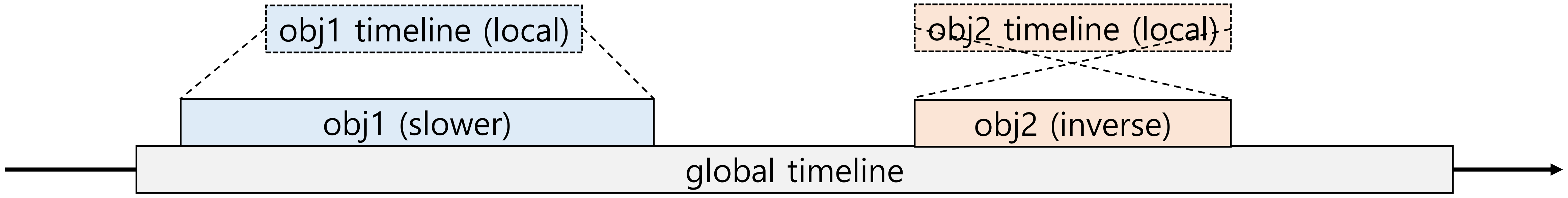
Game Timeline

- We need not limit ourselves to working with the real timeline exclusively.
- We can define a game timeline that is technically independent of real time.
- For example, we can define a game timeline that is independent of real time.
  - If we wish to pause the game, we can simply stop updating the game timeline.
  - If we want our game to go into slow motion, we can update the game clock more slowly than the real time clock.
- When using the approach described above, it is important to realize that the game loop is still running when the game is paused.

Timelines

Mapping Timelines

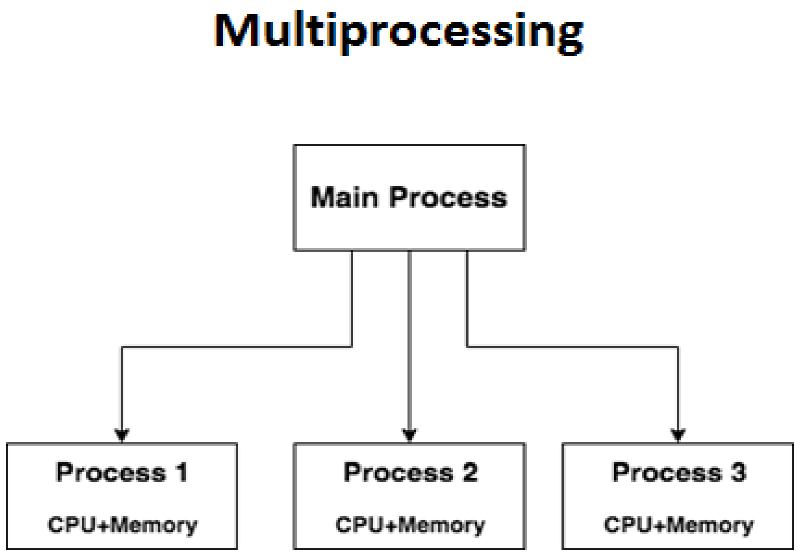
- There is no need to map different timelines at a single rate.
- For example, an animation clip or audio clip might have a local timeline and its play rate is decided when authored or recorded.
- We might want to speed up an animation, or slow down an audio sample.
- We can even play an animation backwards by running its local clock in reverse.
- To enable this, several parameters are involved when performing the mapping between the local timeline and a global timeline.

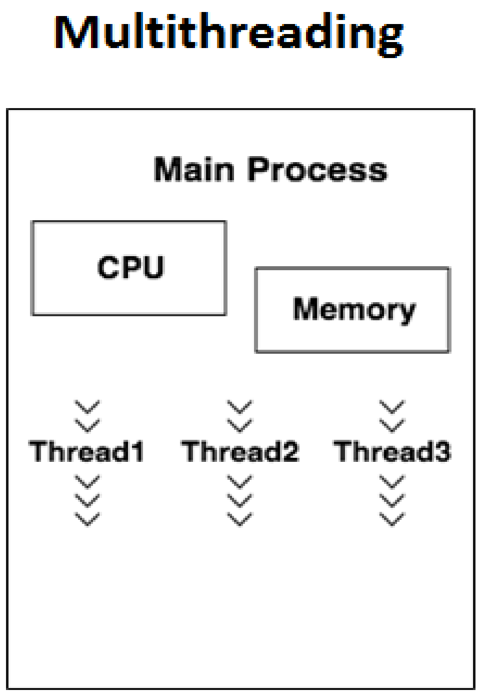


Multithreading

Governing multiple timelines

- Multithreading is the ability of a central processing unit (CPU) to provide multiple threads of execution concurrently, supported by the operating system.
- Governing multiple timelines can be achieved by a multithreading technique.
- Multithreading approach differs from multiprocessing.
  - The threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer.





Multithreading

Advantages and disadvantages of multithreading

- Advantages:
  - If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed.
  - If a thread cannot use all the computing resources of the CPU, running another thread may prevent those resources from becoming idle.

Multithreading

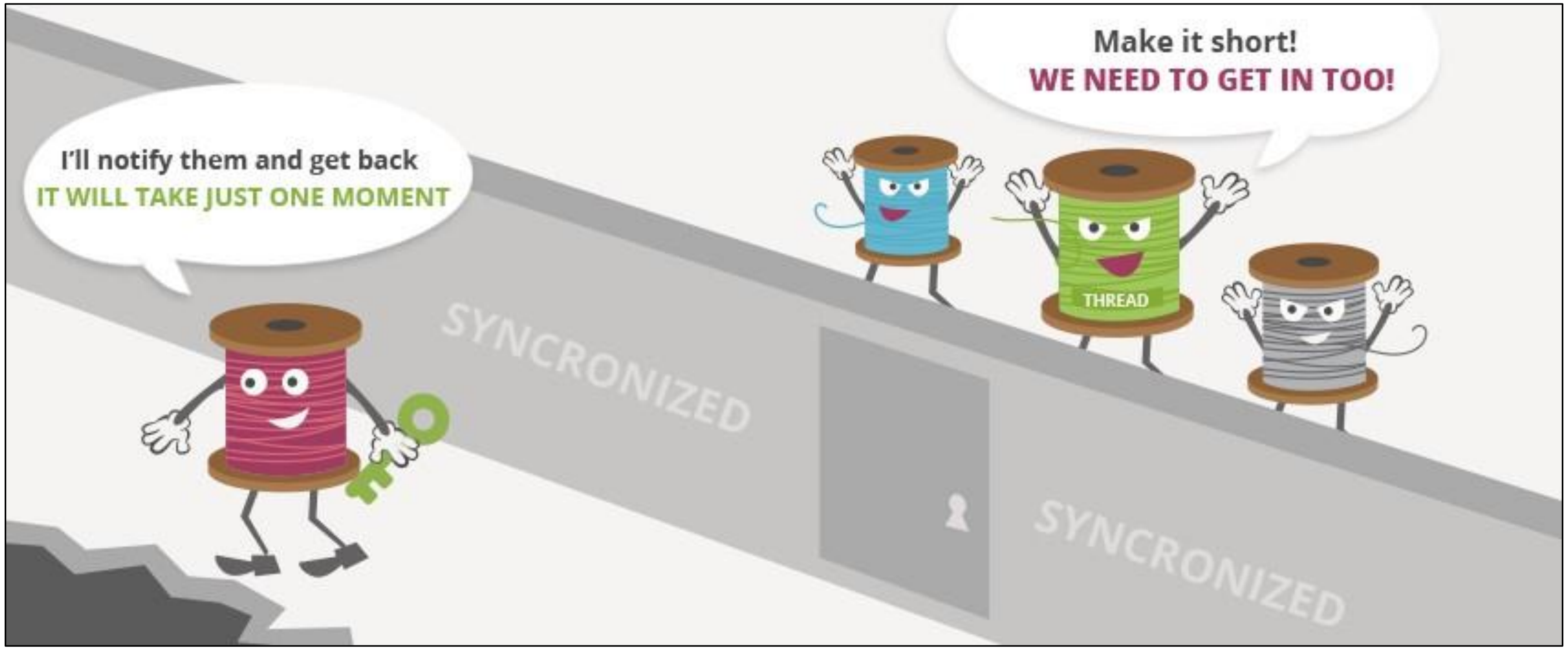
Advantages and disadvantages of multithreading

- Disadvantages:
  - Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.
  - Overall efficiency varies; Hand-tuned assembly language programs using MMX or AltiVec extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.
  - Threads run one at a time in such a way as to provide an illusion of concurrency. Therefore, execution of multiple threads on a single CPU suffers from thread scheduling issue which requires computational resources.

Multithreading

Thread safety

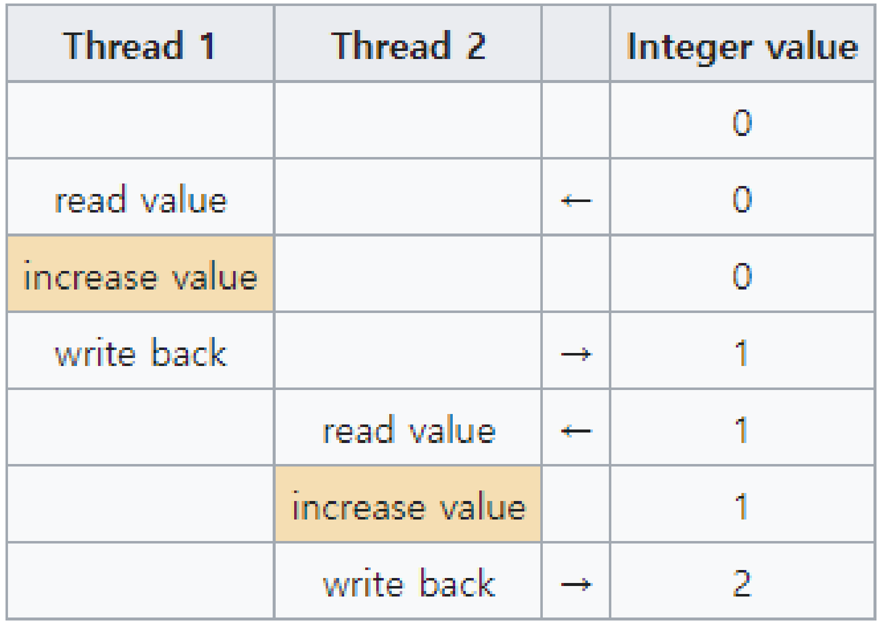
- Thread-safe code ensures all that all threads behave properly and fulfill their design specifications without unintended interaction.
- In other words, it guarantees to be free of race conditions when accessed by multiple threads simultaneously.



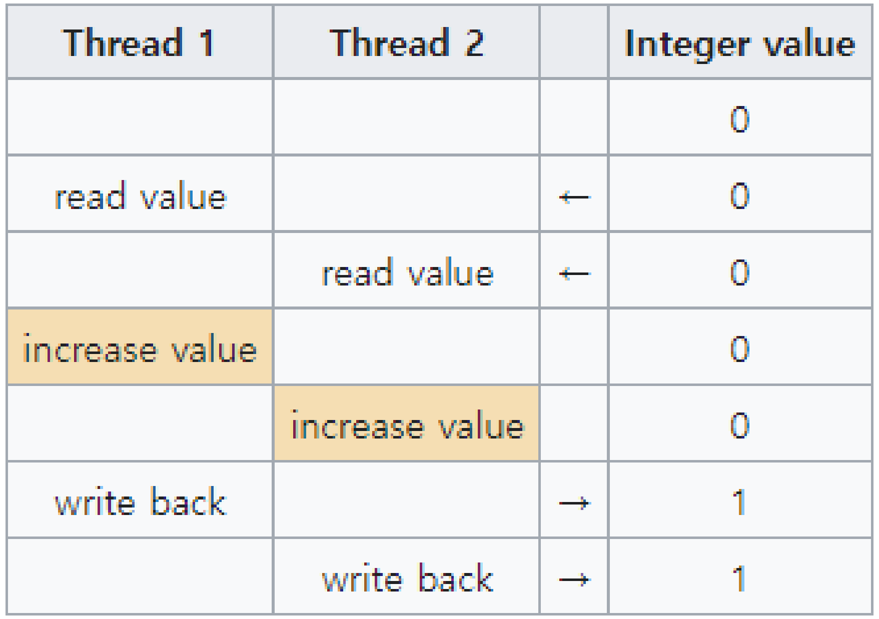
Multithreading

Race condition

- A race condition (or race hazard) arises in software when a computer program depends on the sequence or timing of the program’s processes or threads.
- Critical race conditions often happen when the processes or threads depend on some shared state. Shared states should be done in critical sections that must be mutually exclusive. Failure to obey this rule can corrupt the shared state.



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

In the left figure, two threads increment the value of a global integer and obtain value 2.
However, if the two threads run simultaneously without synchronization, the outcome could be wrong.

Measuring and Dealing with Time

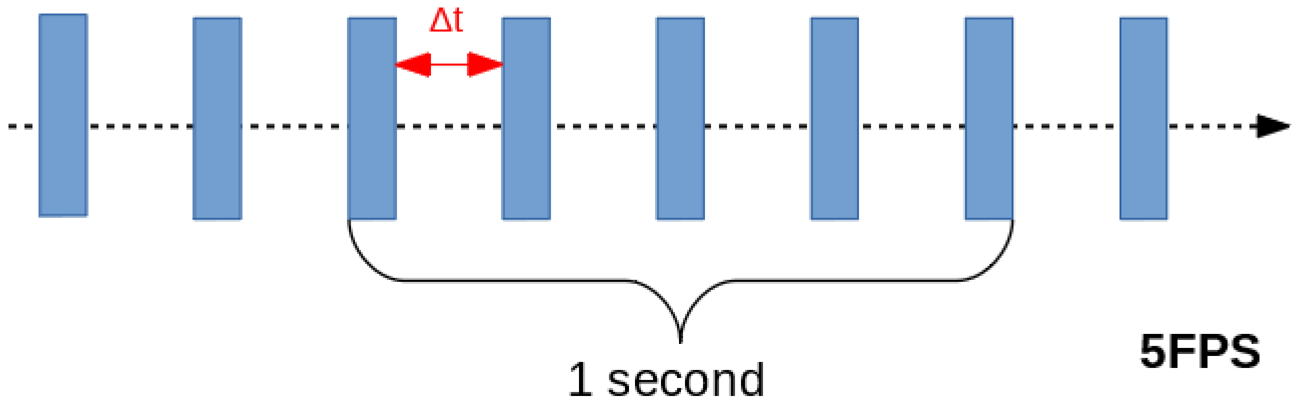
Frame Rate

- The frame rate of a real-time game describes how rapidly the sequence of still 3D frames is presented to the viewer.
- The unit of Hertz (Hz), defined as the number of cycles per second, can be used to describe the rate of any periodic process.
- In games and film, frame rate is typically measured in *frames per second* (FPS).
- Films traditionally run at 24FPS.
- Games are typically rendered 30 ~ 60 FPS.

Measuring and Dealing with Time

Delta Time

- The amount of time that elapses between frames is known as the *frame time*, *time delta*, or *delta time*.
- Delta time is commonplace and it is often represented mathematically by the symbol Δt.
  - If a game is being rendered at exactly 30 FPS, then its *delta time* is 1/30 of a second, or 33.3 ms.
  - If a game is being rendered at exactly 60 FPS, then its *delta time* is 1/60 of a second, or 16.6 ms.



Measuring and Dealing with Time

From Frame Rate to Speed

- Let’s imagine that we want to make a spaceship fly through our game world at a constant speed of 40 meters per second.
- One simple way to accomplish this is to multiply the ship’s speed v by the duration of one frame Δt, yielding a change in position $\Delta x = v\,\Delta t$.
- This delta position can then be added to the ship’s current position $x(t)$, in order to find its position in next frame:
  - $x(t+1) = x(t) + \Delta x = x(t) + v\,\Delta t$
- This is actually a simple form of *numerical integration* known as the *explicit Euler* method.

Question:

- Why do we use Δt instead of frame number?

Measuring and Dealing with Time

CPU-dependent Games

- In many early video games, no attempt was made to measure how much real time had elapsed during the game loop.
- The programmers would essentially ignore Δt altogether and instead specify the speeds of objects directly in terms of meters per frame.
- The net effect of this simplistic approach was that the perceived speeds of the objects in these games were entirely dependent upon the frame rate that the game was actually achieving on a particular piece of hardware.
- If this kind of game were to be run on a computer with a faster CPU than the machine for which it was originally written, the game would appear to be running in fast forward.

Measuring and Dealing with Time

Updating Based on Elapsed Time

- To make games CPU-independent, we must measure Δ*t* in some way.
- A straightforward way is to read the value of the CPU’s high-resolution timer twice - once at the beginning of the frame and once at the end.
- Then we subtract, producing an accurate measure of Δ*t* for the frame that has just passed.

Measuring and Dealing with Time

Updating Based on Elapsed Time

- However, such approach has a big problem.
  - The use of measured value of current frame as an delta time of the upcoming frame is not accurate.
  - Something might happen next frame that causes it to take much more time than the current frame.
  - Using last frame’s delta as an estimate of the upcoming frame can have some very real detrimental effects.

Measuring and Dealing with Time

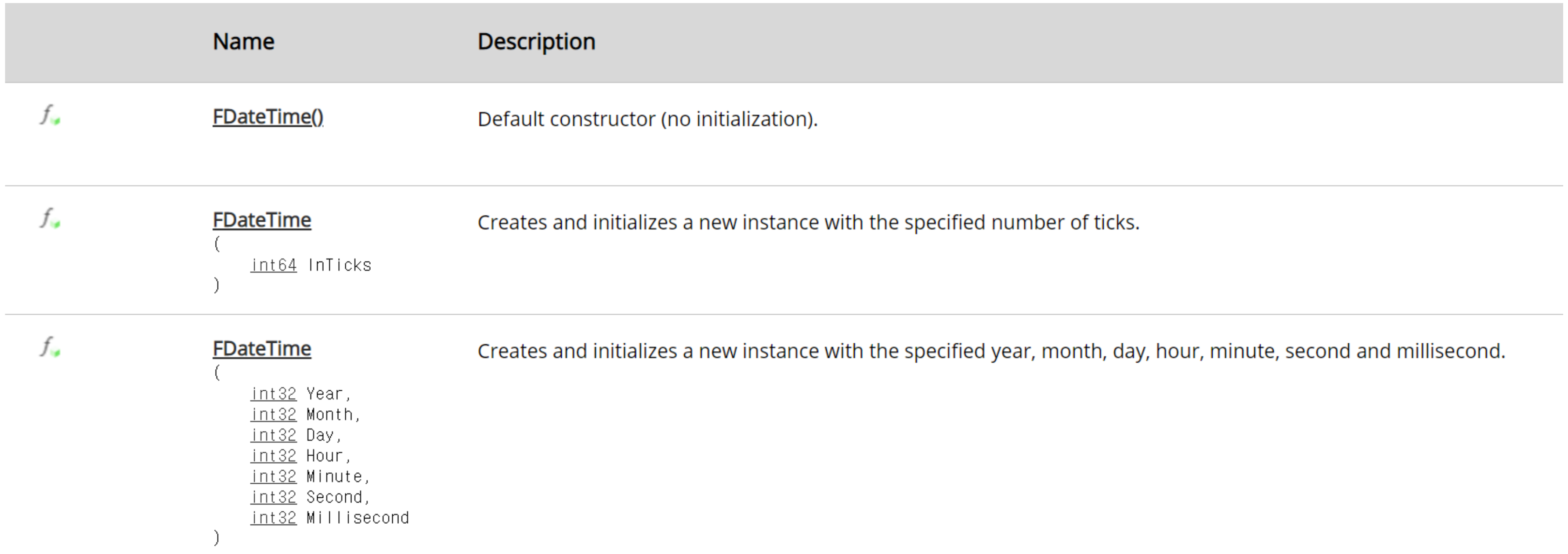
Governing the Frame Rate

- We can avoid the inaccuracy of using last frame's Δt as an estimate of this frame's duration altogether, by flipping the problem in its head.
- Rather than trying to guess at what next frame's duration will be, we can instead attempt to guarantee that every frame's duration will be exactly 33.3 ms (if we are running at 30 FPS).
    - If the measured duration is less than the ideal frame time, we simply put the main thread to sleep until the target frame time has elapsed.
    - If the measured duration is more than the ideal frame time, we can skip the whole frame.
- If the scene contains lots of expensive-to-draw objects on one frame, it will be good choice to draw some of them in next frame.
- Keeping the frame rate consistent can be important.
    - Updating a physics simulation at a constant rate often produces the best result.
    - Features like *record and playback* become a lot more reliable when elapsed frame times are consistent.

Measuring and Dealing with Time

Clock Class

- Some game engines encapsulate their clock variables in a class.
- A clock class typically contains a variable that tracks the absolute time that has elapsed since the clock was created.
- It can also support some nifty features, like time scaling.



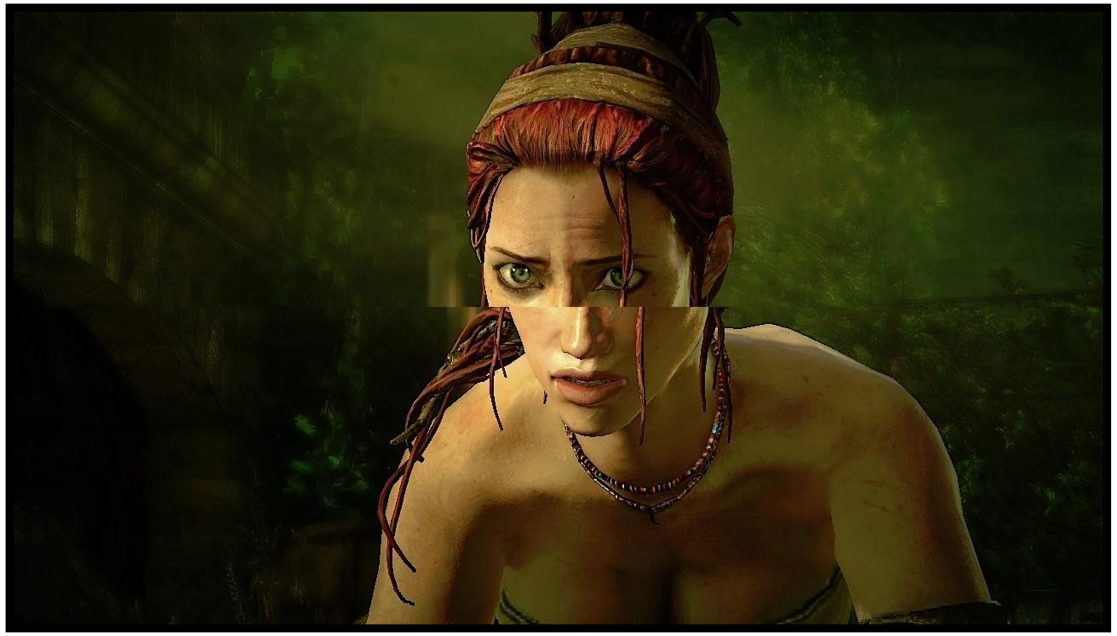
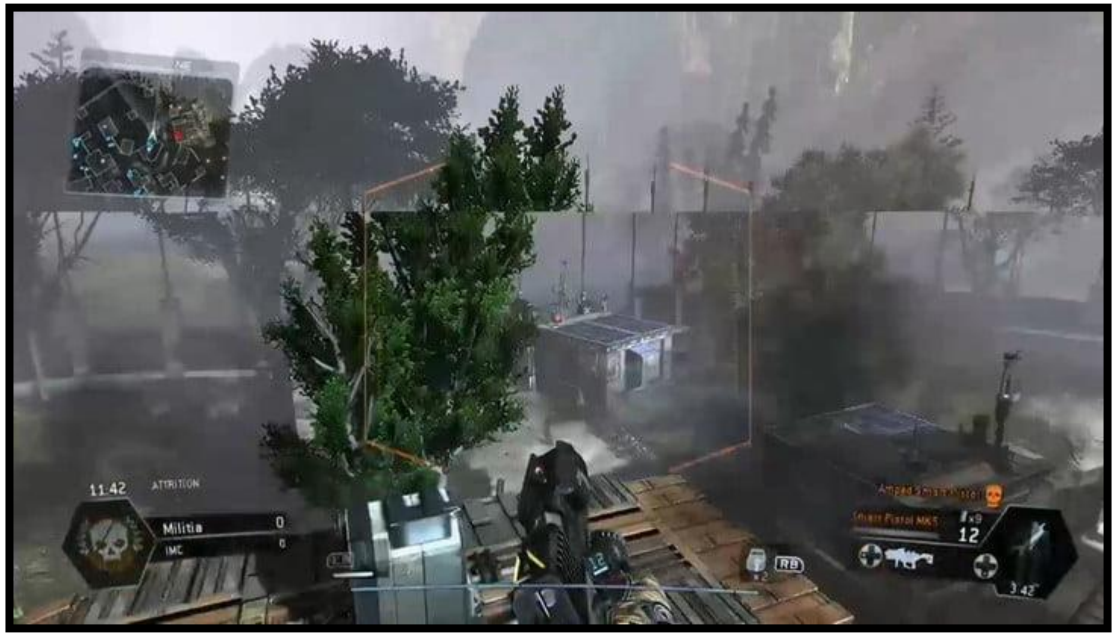
	Name	Description
ƒ	FDateTime()	Default constructor (no initialization).
ƒ	FDateTime (int64 InTicks)	Creates and initializes a new instance with the specified number of ticks.
ƒ	FDateTime (int32 Year, int32 Month, int32 Day, int32 Hour, int32 Minute, int32 Second, int32 Millisecond)	Creates and initializes a new instance with the specified year, month, day, hour, minute, second and millisecond.

UNREAL FDateTime class

V-Sync

Screen Tearing and V-Sync

- A visual anomaly known as *screen tearing* occurs when the back buffer is swapped with the front buffer while the screen has only been partially "drawn" by the video hardware.
- When tearing occurs, a portion of the screen shows the new image, while the remainder shows the old one.

screen tearing examples

V-Sync

Screen Tearing and V-Sync

- To avoid tearing, many rendering engines wait for the *vertical blanking interval* of the monitor before swapping buffers.
- Older CRT monitors and TVs draw the contents of the in-memory frame buffer by exciting phosphors (형광체) on the screen via a beam of electrons that scans from left-to-right and top-to-bottom.
  - On such displays, the v-blank interval is the time during which the electron gun is blanked (turned off) while it is being reset to the top-left corner of the screen.

V-Sync

Screen Tearing and V-Sync

- Modern LCD, plasma and LED displays no longer use an electron bean, and they don’t require any time between finishing the draw of the last scan line of one frame and the first scan line of the next.
  - But the v-blank interval still exists, in part because video standards were established when CRTs were the norm, and in part because of the need to support older displays.
- Waiting for the v-blank interval is called v-sync.
  - This is another form of frame-rate governing, because it effectively clamps the frame rate of the main game loop to a multiple of the screen’s refresh rate.
  - For example, on an NTSC monitor that refreshed at a rate of 60 Hz, the game’s real update rate is effectively quantized to a multiple of 1/60 of a second.
  - If more than 1/60 of a second elapses between frames, we must wait until the next v-blank interval, which means waiting 2/60 of a second.

Networked Multiplayer Game Loops

Client-Server

- In the client-server model, the vast majority of the game's logic runs on a single server machine.
  - Hence the server's code closely resembles that of a non-networked single-player game.
  - Multiple client machines can connect to the server in order to take part in the online game.
  - The client is basically a rendering engine that simply renders whatever the server tells it to render.
  - Great pains are taken in the client code to ensure that the inputs of the local human player are immediately translated into the actions of the player's character on-screen.

Networked Multiplayer Game Loops

Client-Server

- The server may be running on a dedicated machine, in which case we say it is running in dedicated server mode.
  - However, the client and server need not be on separate machines, and in fact it is quite typical for one of the client machines to also be running the server.
  - In fact, the single-player game mode is really just a degenerate multiplayer game, in which there is only one client, and both the client and server are running on the same machine.
  - This is known as *client-on-top-of-server* mode.
- The game loop of a client and server could be implemented as entirely separate processes, since they are conceptually separate entities.
  - However, the client and server need not be on separate machines, and in fact it is quite typical for one of the client machines to also be running the server.
  - As a result, many games run both client and server in a single thread, serviced by a single game loop.

Networked Multiplayer Game Loops

Client-Server

- It’s important to realize that the client and server code can be updated at different rates.
  - For example, in Quake, the server runs at 20 FPS (50 ms per frame), while the client typically runs at 60 FPS (16.6 ms per frame).
  - To implement this, the amount of time that has elapsed since the last server update is tracked, and when it reaches or exceeds 50 ms, a server frame is run and the timer is reset.

Networked Multiplayer Game Loops

Peer-to-Peer

- In the peer-to-peer multiplayer architecture, every machine in the online game acts somewhat like a server and somewhat like a client.
  - One and only one machine has authority over each dynamic object in the game. So, each machine acts like a server for those objects over which it has authority.
  - For all other objects in the game world, the machine acts like a client, rendering the objects in whatever state is provided to it by that object’s remote authority.
  - In a client-server model, it is usually quite clear which code is running on the server and which code is client-side. But in a peer-to-peer architecture, much of the code needs to be set up to handle two possible cases: one in which the local machine has authority over the state of an object in the game, and one in which the object is just a dumb proxy for a remote authoritative representation.