Introduction to Game Architecture

Behrouz Minaei

Iran University of Science and Technology

Real Time Softwares

- Video games are software applications. Specifically, they belong to a class called real-time software applications.

- In a formal definition, real-time software means computer applications that have a time-critical nature or, more generally, applications in which data acquisition and response must be performed under time-constrained conditions.

- Processing and displaying data in a time constrained manner.

Real Time Softwares

As a summary, games are time-dependent interactive applications, consisting of a virtual world simulator that feeds real-time data, a presentation module that displays it, and control mechanisms that allow the player to interact with that world.

Because the interaction rate is fast, there is a limit to what can be simulated. But game programming is about trying to defy that limit and creating something beyond the platform's capabilities both in terms of presentation and simulation. This is the key to game programming

Real Time Loops

As mentioned earlier, all real-time interactive applications consist of three tasks running concurrently.

- First, the state of the world must be constantly recomputed.
- Second, the operator must be allowed to interact with it.
- Third, the resulting state must be presented to the player, using onscreen data, audio, and any other output device available.

In a game, both the world simulation and the player input can be considered tasks belonging to the same global behavior, which is "updating" the world.

In the end, the player is nothing but a special-case game world entity.

Real Time Loops

As soon as we try to lay down these two routines in actual game code, problems begin to appear.

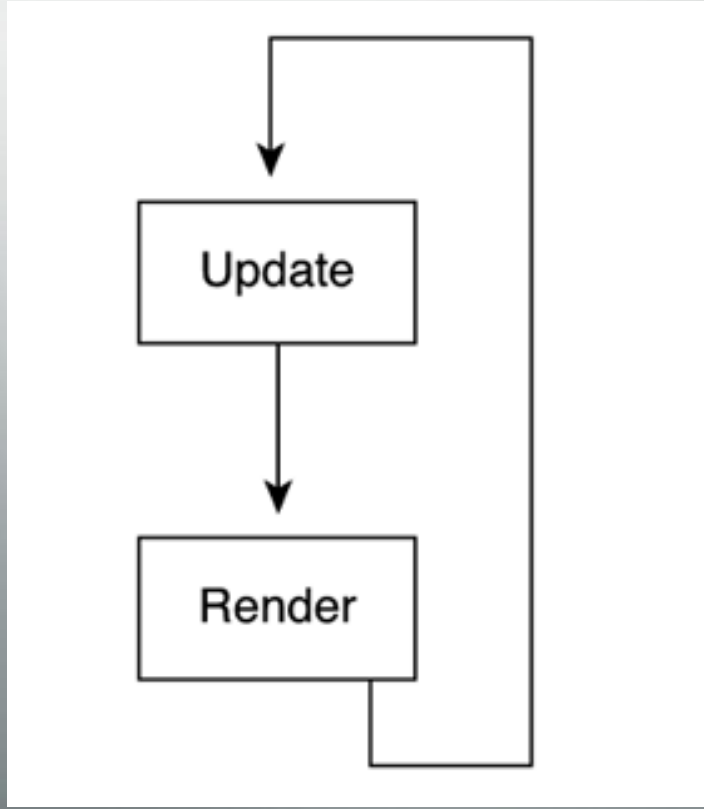
How can we ensure that both run simultaneously, giving the actual illusion of peeking into the real world through a window?

In an ideal world, both the update and render routines would run in an infinitely powerful device consisting of many parallel processors, so both routines would have unlimited access to the hardware's resources.

But real-world technology imposes many limitations:

- Most computers generally consist of only one, two or four processors with limited memory and speed.
- Clearly, the processor can only be running one of the two tasks at any given time, so some clever planning is needed.

Real Time Loops



Real Time Loops

A first approach would be to implement both routines in a loop so each update is followed by a render call, and so forth.

This ensures that both routines are given equal importance.

Logic and presentation are considered to be fully coupled with this approach.

But what happens if the frames-per-second rate varies due to any subtle change in the level of complexity?

Real Time Loops

Imagine a 10 percent variation in the scene complexity that causes the engine to slow down a bit.

Obviously, the number of logic cycles would also vary accordingly.

Even worse, what happens in a PC game where faster machines can outperform older machines by a factor of five?

Will the AI run slower on these less powerful machines?

Clearly, using a coupled approach raises some interesting questions about how the game will be affected by performance variations.

Real Time Loops

To solve these problems, we must analyze the nature of each of the two code components.

Generally speaking, the render part must be executed as often as the hardware platform allows; a newer, faster computer should provide smoother animation, better frame rates, and so on.

But the pacing of the world should not be affected by this speed boost. Characters must still walk at the speed the game was designed for or the gameplay will be destroyed.

Imagine that you purchase a football game, and the action is either too fast or too slow due to the hardware speed.

Clearly, having the render and update sections in sync makes coding complex, because one of them (update) has an inherent fixed frequency and the other does not.

Real Time Loops

One solution to this problem would be to still keep update and render in sync but vary the granularity of the update routine according to the elapsed time between successive calls.

We would compute the elapsed time (in real-time units), so the update portion uses that information to scale the pacing of events, and thus ensure they take place at the right speed regardless of the hardware.

Clearly, update and render would be in a loop, but the granularity of the update portion would depend on the hardware speed—the faster the hardware, the finer the computation within each update call.

Although this can be a valid solution in some specific cases, it is generally worthless.

As speed and frames-per-second increase, it makes no sense to increase the rate at which the world is updated. Does the character AI really need to think 50 times per second? Decision making is a complex process, and executing it more than is strictly needed is throwing away precious clock cycles.

Real Time Loops

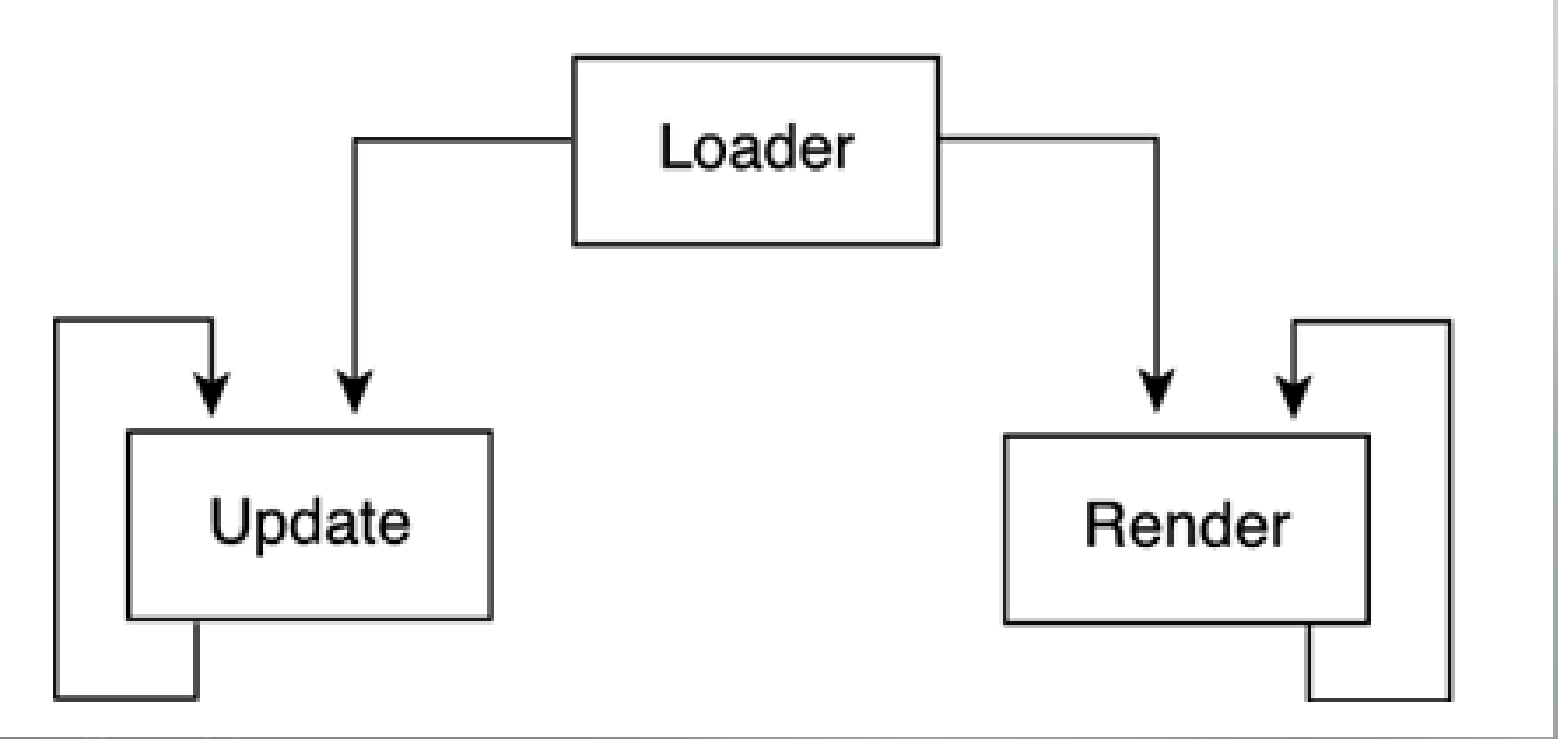
A different solution to the synchronization problem would be to use a twin-threaded approach

so one thread executes the rendering portion while the other takes care of the world updating.

By controlling the frequency at which each routine is called, we can ensure that the rendering portion gets as many calls as possible while keeping a constant, hardware-independent resolution in the world update.

Executing the AI between 10 and 25 times per second is more than enough for most games.

Real Time Loops



Real Time Loops

But the threaded approach has some more serious issues to deal with.

Basically, the idea is very good but does not implement well on some hardware platforms.

Some single-CPU machines are not really that good at handling threads, especially when very precise timing functions are in place. Variations in frequency occur, and the player experience is degraded.

The problem lies not so much in the function call overhead incurred when creating the threads, but in the operating system's timing functions, which are not very precise.

Thus, we must find a workaround that allows us to simulate threads on single-CPU machines.

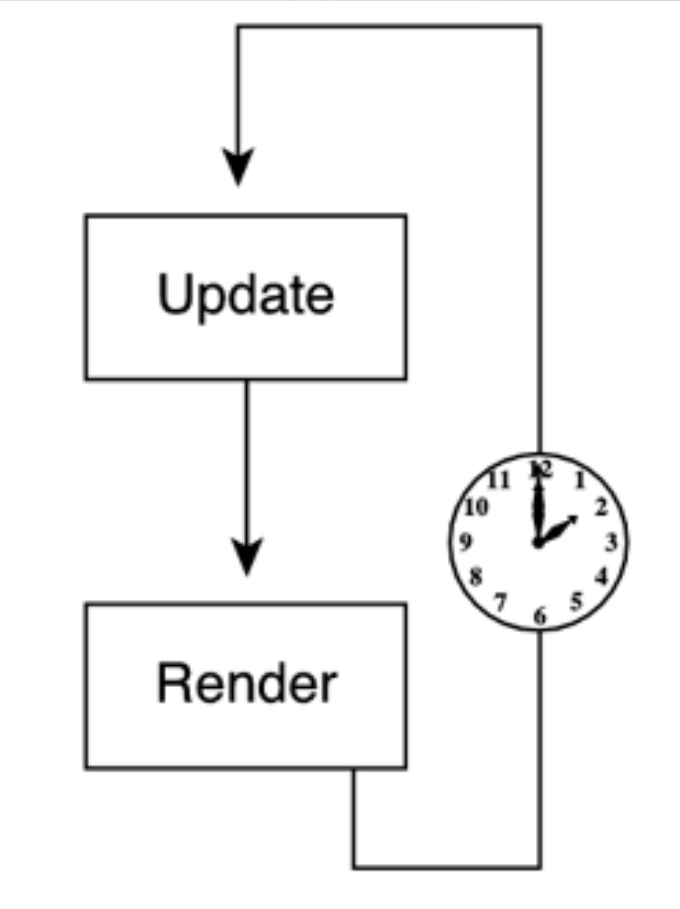
Real Time Loops

The most popular alternative for those platforms that do not support a solid concurrency mechanism is to implement threads using regular software loops and timers in a single-threaded program.

The key idea is to execute update and render calls sequentially, skipping update calls to keep a fixed call rate.

We decouple the render from the update routine. Render is called as often as possible, whereas update is synchronized with time.

Real Time Loops



Final Structure of a real time game loop

```
long timelastcall=timeGetTime();
while (!end)
    {
    if ((timeGetTime()-timelastcall)>1000/frequency)
        {
        game_logic();
        timelastcall=timeGetTime();
        }
    presentation();
    }
```

The Game Logic Section

Every Game Logic section of a game loop probably consists of three sections:

- Player Update
- World Update
- NPC Update

The Game Logic Section- Player Update

A game must execute a routine that keeps an updated snapshot of the player state.

As a first step in the routine, interaction requests by the player must be checked for. (Getting the input)

We will not directly map input to the player's actions because there are some items that can restrict the player's range of actions.

He might indicate that he wants to move forward, but a wall may be blocking his path. Thus, a second routine must be designed that implements restrictions to player interaction.

The Game Logic Section- Player Update

Imagine a game such as Nintendo's classic The Legend of Zelda. The three routines mentioned earlier would have the following responsibilities:

- The "player input" module would effectively read the game controller using specific calls, and then convert the raw data to game world data that makes sense.
- The "player restrictions" routine would access the game world structure because we need to know which level the player is in and what surrounds him or her. This way we can compute both geometrical restrictions, also known as collision detection, and logical restrictions, which basically deal with states the player must be in to be able to perform certain interactions.
- The "player update" routine would map the restrictions to the interactions and generate the right world-level responses. If the player was pressing left and there is no obstacle in that direction, we must trigger the moving animation and update his position, and so on.

The Game Logic Section- Player Update

How about the games that doesn’t have a clear avatar in the screen?

How about Tetris for example?

The same routine applies!

The Game Logic Section- World Update

In addition to the player's action, the world keeps its own agenda, showing activity that is generally what the user responds to.

To begin with, a distinction must be made into two broad game world entities.

- On the one hand, we have passive entities, such as walls and most scenario items. To provide a more formal definition, these are items that belong to the game world but do not have an attached behavior.

- These items play a key role in the player restriction section, but are not very important for the sake of world updating. In some games with large game worlds, the world update routines preselect a subsection of the game world.

- On the other hand we have active elements