Game Architecture

Fundamental of Game Development

Morteza Rajabi – Behrouz Minaei

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 and is the subject of this book.

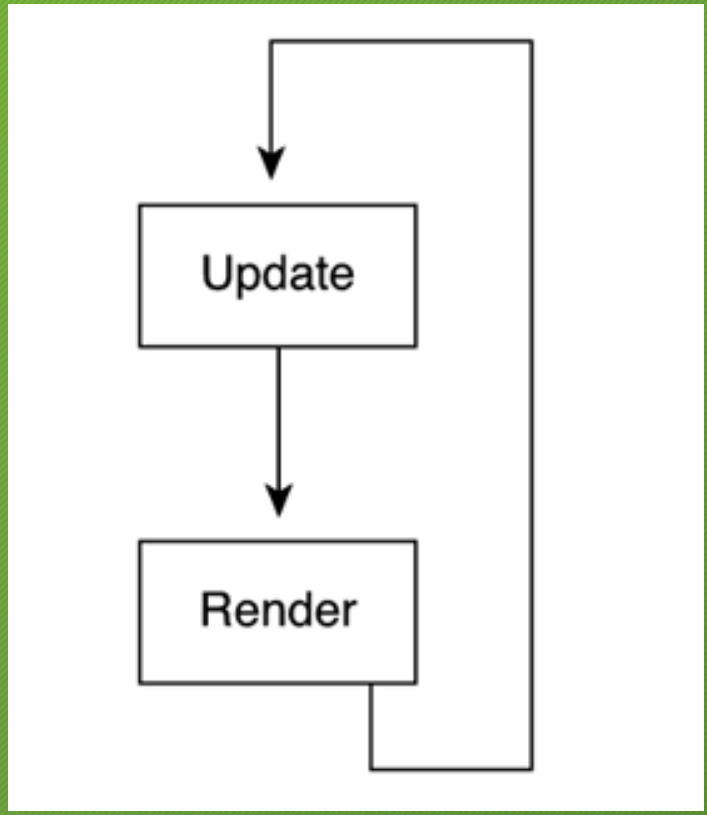
Real Time Loops

- 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.

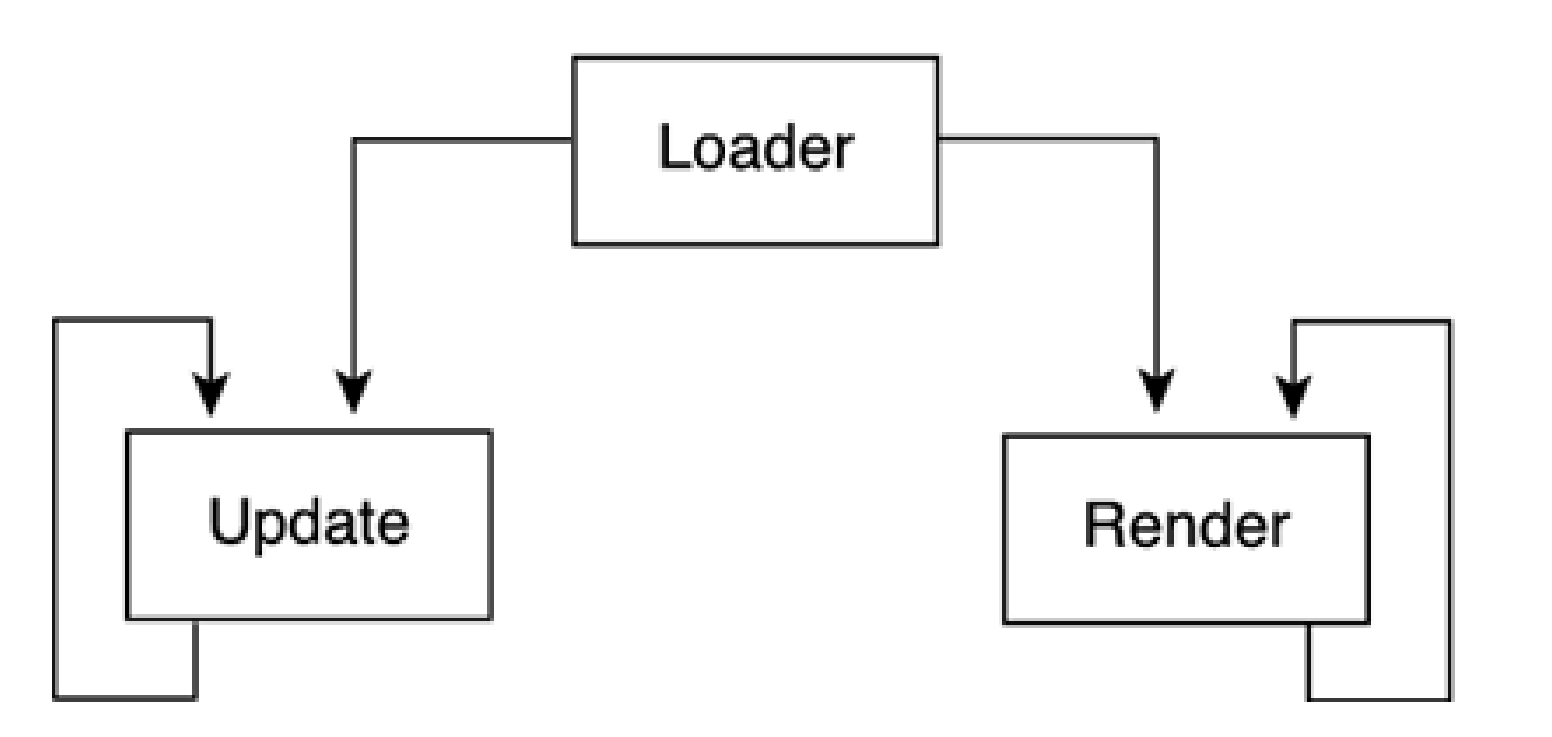
Real Time Loops

- 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



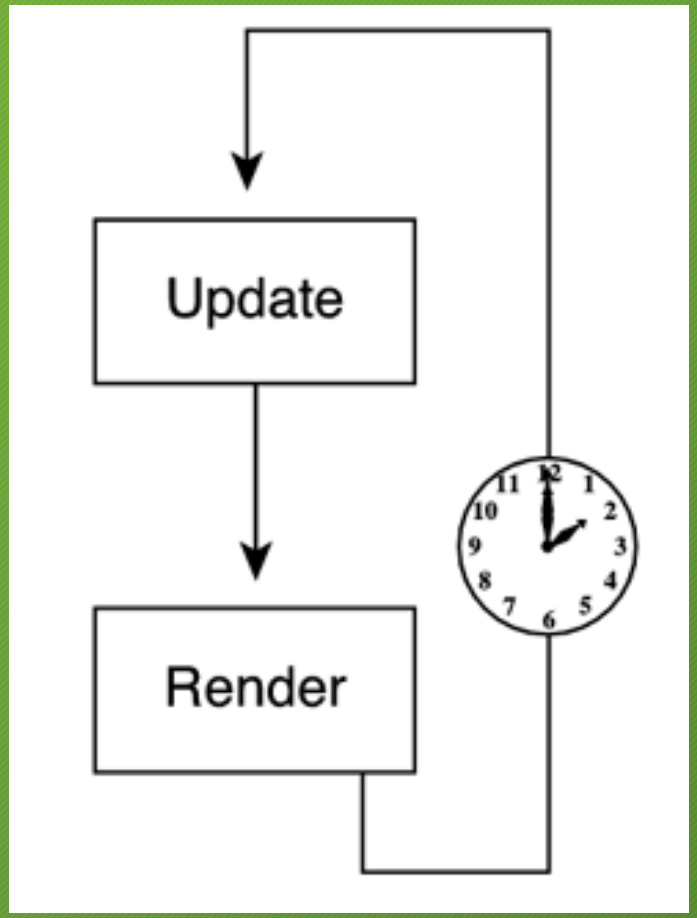
Real Time Loops

- 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 SectionThe 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 UpdateThe 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

The Game Logic Section- World Update

- In our generic game framework, we will assume there is a large number of these active elements, both logic and AI.
- So, the process of updating them will consist of four steps:
  - First, a filter will select those elements that are relevant to the gameplay.
  - An enemy 10 miles away from the player does not seem like a very important item from the player's standpoint, nor is a gate placed in a different game level altogether.
  - This filter must not rule out anything. Some games (like real-time strategy titles, for example) will still need to compute the behavior of all entities.
  - But many times level-of-detail (LOD) techniques will be used for distant items, so having them sorted by relevance is always desirable.

The Game Logic Section- World Update

- Generally, within the overall game framework, AI systems will follow a four-step process too:
  - First, goals and current state must be analyzed. For a flight simulator, this means obtaining the position and heading, state of the weapons systems, and sustained damage for both the AI-controlled and the player-controlled planes. The goal in this case is pretty straightforward: Shoot down the player.
  - Second, restrictions must be sensed. This involves both the logical and geometrical restrictions we already sensed for the player. For our flight simulator example, the main restriction is avoiding a collision with the player and keeping an eye on the ground, so we do not crash into a nearby hill.
- After these two steps, we know everything about our state as AI entities, the player's state, the goal to achieve, and the overall restrictions that apply.

The Game Logic Section- World Update

- Returning to the overall framework, the third step requires that a decision/plan making engine must be implemented that effectively generates behavior rules.
- Fourth, we need to update the world state accordingly. We must store data, such as if the enemy moved, or eliminate it from the data structure if it was shot down by the player. As you will see when we study AI in detail, this four-step process blends extraordinarily well with most game AIs.

The Game Logic Section- World Update

```
Player update
   Sense Player input
   Compute restrictions
   Update player state
World update
   Passive elements
      Pre-select active zone for engine use
   Logic-based elements
      Sort according to relevance
      Execute control mechanism
      Update state
   AI based elements
      Sort according to relevance
      Sense internal state and goals
      Sense restrictions
      Decision engine
      Update world
End
```

The Presentation Section

- This section contains three sections:
  - World Rendering
  - NPC Rendering
  - Player Rendering

The Presentation Section – World Rendering

- Rendering complete game worlds in real-time is almost impossible except for simple games like Tetris.
- For any involved title, some filtering must be applied prior to the main render call to decide what should be taken into consideration and what shouldn't.
- For example, very distant or invisible zones of the world can probably be culled away because they provide little or no information to the player and would just decrease the frame rate due to the added complexity.
- So, any world-rendering pipeline will more or less consist of two parts:
    - selecting the relevant subset
    - taking care of the actual rendering.

The Presentation Section – World Rendering

- For the graphics pipeline, the selection routine is implemented via clipping, culling, and computing occlusions, so the resulting representation is just the visible part of the game world from the player's viewpoint.
- This way we can focus on drawing what actually matters to the player.
- An optional, auxiliary process is sometimes applied to the visible data, which computes the relevance of the data and chooses a suitable level of detail to render it.
- A tree that is seen 500 meters away probably doesn't need 10,000 triangles because each triangle would occupy a tiny fraction of a single pixel. Thus, a low resolution, more efficient representation can be used instead without the detrimental effect on performance.

The Presentation Section – NPC Rendering

- Rendering NPCs is quite different from rendering inanimate geometry.
- They need a specific pipeline due to their animation properties.
- However, we can still begin by filtering the character lists, so only characters close to the player and affecting him are processed.
- Again, a visibility step is pretty common. Only characters that survive a clipping and occlusion test will be moved on to the pipeline.
- This is especially important for fully animated characters. Working with dynamic, animated geometry is more expensive than static, passive elements, and thus must be applied only when needed. Optionally, some games will use an LOD pass to create a simplified representation of characters located in view but far away from the viewer.

The Presentation Section – Player Rendering

- The main player is nothing but a very special-case NPC.
- But its rendering pipeline is simpler than those of secondary characters for two very simple reasons.
  - First, the player is generally visible, so there is no need to allocate time to check him for visibility. After all, he's supposed to be the hero, so it makes no sense for him to remain hidden.
  - Second, there is no need for LOD processing as well. Most games display players in a central role, so they will always use high-resolution meshes.
- Thus, the main player will only undergo an animation step, packing, and a render step.

The final Structure of presentation section

```
World presentation
   Select visible subset (graphics)
      Clip
      Cull
      Occlude
   Select resolution
   Pack geometry
   Render world geometry
   Select audible sound sources (sound)
      Pack audio data
      Send to audio hardware
NPC presentation
   Select visible subset
   Animate
   Pack
   Render NPC data
Player presentation
   Animate
   Pack
   Render
```

Final Game Loop

```
Game logic
   Player update
      Sense Player input
      Compute restrictions
      Update player state
   World update
      Passive elements
         Pre-select active zone for engine use
      Logic-based elements
         Sort according to relevance
         Execute control mechanism
         Update state
      AI based elements
         Sort according to relevance
         Sense internal state and goals
         Sense restrictions
         Decision engine
         Update world
End

Presentation
   World presentation
      Select visible subset
         Clip
         Cull
         Occlude
      Select resolution
      Pack geometry
      Render world geometry
      Select audible sound sources
      Pack audio data
      Send to audio hardware
   NPC presentation
      Select visible subset
      Animate
      Pack
      Render NPC data
   Player presentation
      Animate
      Pack
      Render
End
```