Production Systems and Information Engineering
Volume 11 (3), pp. 51–63
doi: 10.32968/psaie.2023.3.4

GAME LOOP: THE HEART OF THE GAME ENGINE

Péter Mileff
University of Miskolc, Hungary
Department of Information Engineering
mileff@iit.uni-miskolc.hu

Abstract. The world of gaming has come a long way since the days of Pong and Space Invaders. Today, games are not only a form of entertainment, but also a major industry that drives technological advancements in multiple fields. A game is a complex system, consisting of many components. In this publication, we explore one of the most important part of it, the concept of game loops and their role in game technology. A game loop is a fundamental structure in game development that controls the flow of a game by constantly updating the game state and rendering the graphics. This publication aims to provide a comprehensive understanding of various types of game loops and their importance in game technology.

Keywords: Game loop, elapsed time, optimization

1. Introduction

Computer games have become an important form of entertainment and a major part of popular culture. They offer a wide range of benefits and have become an important medium for storytelling, art, and education. Game technology is a rapidly evolving field that encompasses the various tools and techniques used to create and run video games. It includes everything from the programming languages and development tools used to create the game, to the hardware and software platforms that run it. With the increasing popularity of video games and the growth of the gaming industry, game technology has become an essential area of research and development.

Recent advances in game technology have led to a wide range of new possibilities for game developers. From improved graphics and more realistic physics, to new forms of player interaction and more immersive game worlds. The use of machine learning, artificial intelligence and virtual and augmented reality, have also opened up new opportunities for game developers to create more

realistic and engaging experiences for players. Multiplayer games have also become increasingly popular, allowing players from all over the world to connect and compete with one another in real time.

As the gaming industry continues to grow and evolve, game technology will play an increasingly important role in shaping the future of the medium. Whether it's through the development of new hardware and software platforms, or the creation of new forms of player interaction and game design, game technology will continue to drive innovation in the gaming industry. The aim of this publication is to provide an overview about a very important component of a game. We will discuss the importance of game loops, the types of game loops and their advantages and disadvantags.

2. The Game loop

A game is a complex system that involves many interconnected components working together to create a cohesive and enjoyable experience for the player. To make this possible, developers create game engines.

A game engine is a complex system that serves as the foundation for the development and operation of video games. It is a collection of software components, tools, and libraries that provide developers with the necessary functionality to create, test, and deploy games. The engine takes care of many of the technical aspects of game development, such as rendering graphics, managing memory, and handling input and output.

One of the key components of a game engine is the game loop, which controls the flow of the game in real-time. It is responsible for updating the game state, handling player input, and rendering the game to the screen. The game loop is typically implemented as an infinite loop that runs continuously while the game is in progress. It repeatedly processes input, updates game state, and renders the game, providing the illusion of smooth animation and interaction. The game loop is a fundamental concept in game development and is present in nearly all video games. There are several types of game loops that are commonly used in game development. Each type has its own advantages and disadvantages, and the choice of which game loop to use depends on the specific requirements of the game and the platform it is being developed for.

The most known type of loops:

1. **The fixed-time step loop:** This type of game loop runs at a fixed interval, regardless of the amount of time that has passed since the last update. This ensures that the game runs at the same speed on all devices and reduces the chance of bugs caused by variable frame rates. But can

lead to issues if the game becomes too complex or if the player's device is not powerful enough to handle the fixed frame rate.

2. **The variable-time step loop:** This type of game loop updates the game state based on the amount of time that has passed since the last update. This can help to ensure that the game runs smoothly on a wide range of devices, but can lead to issues if the frame rate drops too low.
3. **The semi-fixed time step loop:** This type of game loop combines elements of both fixed-time and variable-time step loops. It uses a fixed time step for the game logic, but adjusts the rendering based on the amount of time that has passed since the last update.
4. **The event-based game loop:** This type of game loop does not run continuously, instead, it waits for specific events to occur (i.e. user input or an update from the server) and then processes them. This approach can be useful for games that require a lower update rate.
5. **The Asynchronous game loop:** This type of game loop allows the game to update and render in an asynchronous fashion, meaning that the game can continue to update and render even while waiting for input or other resources to load. This approach can help to reduce the perceived latency and make the game feel more responsive.

Each iteration of the game loop is known as a frame. Most real-time games update several times per second: 30 and 60 are the two most common intervals. If a game runs at 60 FPS (frames per second), this means that the game loop completes 60 iterations every second.

2.1. Game loop parts

A game loop (or main loop) is typically an infinite loop. A traditional game loop can be broken up into the following parts:

1. **Input handling:** The game loop starts by handling any input from the player, such as keyboard, mouse, or gamepad input. This step is responsible for updating the game state based on the player's actions.
2. **Updating game state:** The game state is updated based on the input received in the previous step, as well as any other factors that affect the game, such as physics and AI. This step is responsible for updating the position of characters, checking for collisions, and updating the game's internal logic.
3. **Rendering graphics:** The game loop then renders the updated game state to the screen. This step is responsible for drawing the characters, objects, and environments to the screen.
4. **Wait/Sleep:** After the graphics have been rendered, the game loop waits for a specific amount of time, known as the frame rate, before

starting the next iteration. This helps to ensure that the game runs at a consistent speed on different devices.

The game loop then repeats these steps, over and over again, until the game is closed. It's worth noting that the order of the steps can vary depending on the game, and on the game engine used. For example, some game engines may handle physics and collision detection on a separate thread, this is known as multi-threading.

2.1.1. The problem with the game loop

From a programming point of view, the basic game loop can be described as follows:

```
while game is running
     Process inputs
     Update game world
     Render world
Loop
```

By filling the loop with appropriate content, our software will be functional. Even before we think that the game loop is so simple, unfortunately, this approach has a very important flaw. Due to the nature of the hardware and the operating system, the loop will run at different performance on different computers. The loop is performed faster on fast machines and slower on slow machines [10]. As a result, the game speed will not be the same.

Consider the following movement as an example:

```
newpos(x,y) = currentpos(x,y) + velocity(v)*direction(x,y)
```

During movement, we perform the above operation for every object in every frame in the game loop, so the movement will be continuous. However, for a slow machine, the speed of movement will be slow, and for a fast computer, it will be too fast. This was a phenomenon in very early games (especially in the DOS era). So there is a need for better approaches that solve this problem.

3. Game loop types

Today's software therefore prefers a modified version of the simple solution outlined above, Time Based Movement, or some modified version of this technique. The algorithm, as its name suggests, starts from the dimension of time. Its basic idea is the following: if we are able to measure the time elapsed between two executed game loops with a higher resolution clock (at least milliseconds), we will obtain a factor that can be used to standardize the speed

between machines. This ensures that the objects move at the same speed even on machines with different speeds.

It's an important value for game loops and it is used in many calculations and decisions. It's a common practice to use elapsed time to make the game updates and animation independent from the frame rate and the device performance. This ensures a consistent game experience on different devices and platforms. For example, if the game's physics are calculated using a fixed time step, the elapsed time is used to determine how far objects should move and how they should respond to forces and collisions.

Elapsed time is ideally a double-precision floating-point number between 0.0 and 1.0. Since some time certainly passes between two consecutive frames, we cannot get a negative value. If the value of the elapsed time is zero, then the resolution of the timer used is not high enough. This means that we need to use a more accurate operating system level clock device for the game implementation that can also measure smaller changes.

In addition, elapsed time can be used to implement time-based effects, such as slowing down or speeding up time in the game.

3.1. Fixed-time step loop

Fixed-time step loops are a type of game loop that provide a consistent and predictable update rate for game logic. In this approach, the game loop operates at a fixed interval, such as 60 times per second. This ensures that the game logic and physics calculations are performed at a consistent rate, providing a stable and predictable gaming experience. For example we want to fix the frames per second (FPS) in our game at 60. In practice, 60 FPS means that approximately 16 milliseconds are available to execute the game loop (1000 ms / 60 = 16.6667 ms). The operation of the algorithm is as follows: if the game loop is able to complete its task within this time interval, or even in less time, then as the last step of the game loop it will wait for exactly the time required to reach 16.6667 ms. As long as we can reliably do all of the game processing and rendering in less than that time, we can run at a steady frame rate.

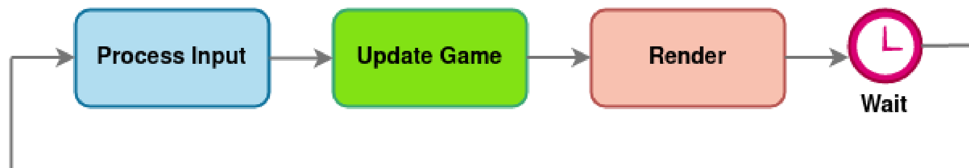

Figure 1. Fixed-time step loop

The following code shows a sample game loop working with this algorithm:

```
double MS_PER_UPDATE = 1.0 / 60.0;
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();
  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

One of the advantages of using a fixed-time step loop is that it helps to eliminate inconsistencies and fluctuations in the game's performance. This is particularly important for games that require precise timing and interactions, such as physics-based games or fast-paced action games. A fixed-time step loop ensures that all calculations are performed consistently, even if the frame rate varies, ensuring a smooth and stable gaming experience.

Another advantage of fixed-time step loops is that they can make it easier to develop and test the game logic. Developers can rely on a predictable update rate, making it easier to debug and refine the game mechanics. This can be especially useful for developing games that require precise timing or interactions.

It's important to note that fixed-time step loops can also have some drawbacks. If the fixed interval is too short, the game may become too demanding for older or less powerful devices, leading to a lower frame rate and less enjoyable experience. In addition, if the fixed interval is too long, the game may become less responsive to player inputs, leading to a less enjoyable experience.

3.2. The variable-time step loop

A variable-time step game loop is a design pattern used in video game development to control the update and render cycles of a game. This loop does not use waiting as the last step of the loop. Instead it applies the elapsed time concept as a measured factor to update the whole game logic. This method helps to prevent "slow motion" or "fast forward" effects when the game is running on slower or faster devices. The longer the frame takes, the bigger steps the game takes. The logical steps of the loop can be described as follows:

```
double lastTime = getCurrentTime();
while (game_is_running)
{
```

```
    double current = getCurrentTime();
    double elapsed_time = current - lastTime;
    processInput();
    update(elapsed_time);
    render();
    lastTime = current;
}
```

In each frame, we determine how much real time passed since the last game update (`elapsed_time`). When we update the game state, we use this time factor to calculate every type of movement in the game.

Consider the same movement equation, but now extended with the new elapsed time factor:

```
newpos(x,y) = currentpos(x,y) + elapsed_time*velocity(v)*direction(x,y)
```

`Elapsed_time` is used as a multiplying factor and therefore a zero value cannot be used. On faster machines, this time will be smaller, because the loop is executed faster due to the higher computational capacity, and on slower machines, we get a higher number value. So, this factor can therefore compensate for the speed difference between the computers. Consider the following example:

In the game, a projectile is shooted that travels across the screen. On slower computers, the value of `newpos(x,y)` will be a higher number due to the bigger `elapsed_time`. In practice, this means that the projectile jumps up to several pixels. Of course, this is not a problem, because the dynamics of the game cover it up, and it only becomes confusing for the human eye when the position distance between the two phases of the projectile is too large. Whereas on a fast computer, due to the small value of `elapsed_time`, the projectile moves in smaller jumps, even smaller than a pixel (that's why coordinates are usually stored in float type). In this case, the movement and gameplay will be very smooth. Overall, the projectile travels the (almost) the same distance in both cases, just not on the same scale.

The disadvantage of this type of game loop is that the game can be non-deterministic. This means that on a new PC, the physics engine updates the projectile's position 50 times, but on an old PC only does it ten times. Most games use floating point numbers, and those are subject to rounding error. Each time we add two floating point numbers, the answer you get back can be a bit off. The fast computer does five times as many operations, so therefore it will accumulate a bigger error than the old PC. The result is: the same projectile will end up in different places on their machines. The precise calculation of collisions thus becomes significantly more difficult.

Besides all that, the variable-time step game loop can greatly improve the overall responsiveness and control feel of the game, especially in games that require precise and responsive game play, such as platformers or first-person shooters.

The variable-step loop solution is almost suitable for our graphics application. But it is advisable to add two more small adjustments.

3.2.1. Correction of the elapsed time

So far, the above solution only deals with the ideal case where nothing interferes with the gameplay. However, in practice, many disturbing factors can occur, for example the computer lagging. During the gameplay, sometimes certain background processes in the operating system can use more resources (for example, we compress in the background, or the antivirus performs some kind of background check, etc.), so the elapsed time can increase many times, which means that the objects perform much larger movements, more pixels in one loop. Another typical example of this is debugging. When the software is stopped for debugging purposes, and then restarted, the value of the elapsed time will be very high, since the time measured in the previous frame is far away from the current one. This behavior, the needle-like spikes in the `elapsed_time` results in a non-smooth gameplay.

It is therefore advisable to set an upper limit for the value, for example a value of 1.0.

```
if (elapsed_time > 1.0f) {
        elapsed_time = 1.0f;
}
```

With this solution, although it is possible to maximize the peaks of the `elapsed_time` values, the sudden change in the `elapsed_time` value over a wide interval due to the load of the computer is still a problem. These changes can be seen in the gameplay. In the case of an FPS game, even the movement in space can be "trembling". Another very useful optimization of the variable-time step loop is the "smoothing" of the `elapsed_time`. A very simple but well-functioning solution is to calculate the value with which the game state will be updated from the average of the current and previous `elapsed_time` values.

```
elapsed_time += curr_frame_tick - prev_frame_tick;
elapsed_time *= 0.5;
```

Although it is not possible to eliminate random loads on the computer, their effects can be corrected by this method. The result is a much smoother gameplay.

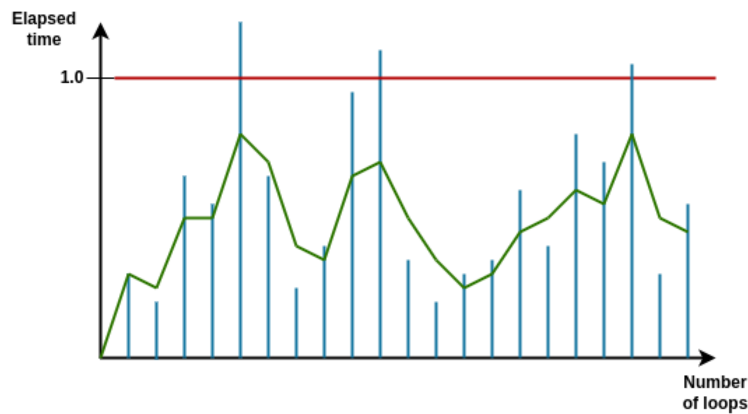

Figure 2. Elapsed time smoothing

Figure 2. shows a sample possible series of the elapsed time values. The red line is the upper limit and the green value is the average of the current and the previous time.

3.3. Semi-fixed timestep

The semi-fixed timestep game loop is a commonly used method for controlling the simulation of physics and other real-time processes in video games. It combines the benefits of both fixed timestep and variable timestep approaches to create a balance between stability and efficiency. The algorithm uses a fixed timestep for most of the simulation updates, but allows for variable timestep updates in certain cases, such as when the game is running slowly and needs to catch up. This allows for smoother and more consistent simulations, while also allowing for more efficient use of processing resources.

The semi-fixed timestep game loop works as follows:

1. Initialization: Initialize the game state and variables, including the time step (dt) and the accumulator (acc).
2. Input handling: Process user input, update the game state accordingly.
3. Fixed timestep simulation: The game state is updated using a fixed timestep (dt), with the number of simulation steps determined by the accumulator (acc).
4. Variable timestep simulation: If the accumulator (acc) is larger than the fixed timestep (dt), additional simulation steps are taken using a variable timestep until the accumulator is reduced to a value less than dt.

5. Render: The game state is rendered to the screen, using the updated simulation data.
6. Repeat: Repeat the game loop until the game is terminated.

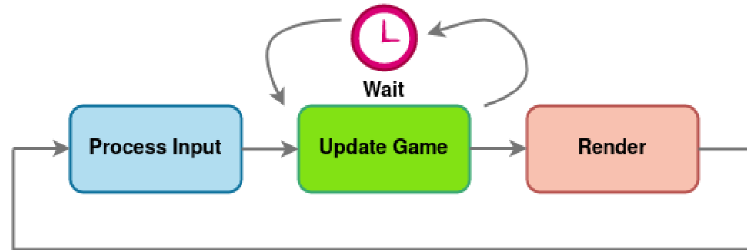

Figure 3. The semi-fixed timestep loop

The loop can be described as follows:

```
double previous = getCurrentTime();
double frameTime = 0.0;
double MS_PER_UPDATE = 1.0 / 60.0;
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  frameTime += elapsed;

  processInput();

  while (frameTime >= MS_PER_UPDATE)
  {
    update();
    frameTime -= MS_PER_UPDATE;
  }

  render();
}
```

At the beginning of each frame, we update *frameTime* based on how much real time passed. This measures how far the game's clock is behind compared to the real world. After that we process the inputs and then we reach the inner loop whose task is to update the game state, one fixed step at a time, until the *frameTime* is bigger than the `MS_PER_UPDATE`. Finally we render and start

over again. The shorter the MS_PER_UPDATE is, the more processing time it takes to catch up to real time. The longer it is, the choppier the gameplay is.

3.3.1. *The residual lag problem*

Residual lag refers to the discrepancy between updating the game at regular fixed time steps while rendering occurs at arbitrary time points. Consequently, from the user's standpoint, the game frequently appears to display at a time between two consecutive updates.

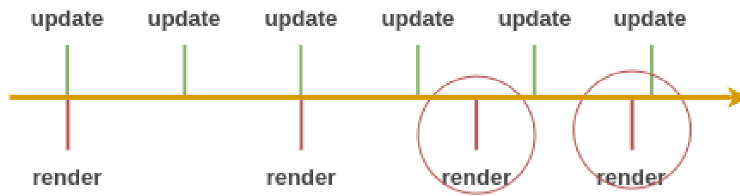

Figure 4. The residual lag problem

Although the update phase occurs at regular intervals, the render phase lacks determinism. Its occurrence is less frequent than updates and it is not steady either. The challenge lies in the fact that rendering does not always coincide precisely with the point of updating. In Figure 4, the instances where this discrepancy occurs are denoted by red circles.

Consider the following scenario: A moving object traverses across the screen. During the first update, the object is on the left side, and the subsequent update places it on the right side. When the game is rendered at a time between these two updates, users naturally expect to see the object positioned in the center of the screen. However, due to the current solution, the object remains on the left side, causing the motion to appear jagged or stuttery.

Fortunately, we possess precise information about the intermediate position between update frames at the time of rendering, stored as "frameTime." Instead of stopping the update loop when frameTime reaches zero, we stop it when it becomes less than the update time step. This approach leaves us with a leftover time amount, representing how far we are into the next frame. As the rendering phase commences, we can utilize this time information effectively:

```
render(frameTime / MS_PER_UPDATE);
```

We divide by MS_PER_UPDATE here to normalize the value. The value passed to *render()* will vary from 0 (right at the previous frame) to just under 1.0 (right

at the next frame), regardless of the update time step. This way, the renderer doesn't have to worry about the frame rate. It just deals in values from 0 to 1.

The renderer possesses knowledge of every game object and its present velocity. Let's consider a hypothetical scenario where an object is positioned 100 pixels from the left side of the screen and is moving to the right at a speed of 400 pixels per frame. In the situation where we find ourselves precisely midway between frames, the value 0.5 is passed to the *render()* function. Consequently, the object is drawn half a frame ahead, resulting in its appearance at 250 pixels from the starting point.

Certainly, sometimes the extrapolation may prove to be inaccurate. When computing the next frame, it's possible to discover that the object encountered an obstacle, decelerated, or experienced other factors affecting its motion. During rendering, we interpolate the object's position between its last known position and the anticipated position on the next frame. However, we cannot confirm the accuracy of this interpolation until we complete the full update, incorporating physics and AI calculations. As a result, the extrapolation occasionally yields partially incorrect values. Fortunately, these inaccuracies typically go unnoticed or have minimal impact, especially when compared to the jarring visual disturbances that arise when no extrapolation is employed.

4. Conclusion

Game loops play a crucial role in the design and development of video games. They provide the foundation for gameplay and ensure a consistent and enjoyable player experience. Game loops consist of various elements, including input, rules, feedback, and progression, which work together to create a cycle of interaction and challenge. A well-designed game loop keeps players motivated to continue playing by providing a sense of progression, accomplishment, and reward. This paper made an overview about the most common type of game loops. Advantages and disadvantages were presented. Moreover some optimization tips were also discussed. Understanding the importance of game loops can help game developers to create engaging and satisfying games.

References

[1] Akenine-möller, T., Haines, E.: *Real-Time Rendering.* A. K. Peters. 3nd Edition, 2008.

[2] João M. P. Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz: *Embedded Computing for High Performance*, Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, pp. 17-56., 2017.

[3] Charles Kelly.: *Programming 2D Games*, A K Peters/CRC Press; 1st edition, 2012.

[4] Jason Gregory: *Game Engine Architecture*, A K Peters/CRC Press; 3rd edition, 2018.

[5] MARÍN-LORA, C.; CHOVER, M.; REBOLLO, C., REMOLAR, I.: *A game development environment to make 2D games*, Communication Papers, Vol.9 – No18, pp. 7-23, 2020.

[6] Pitt, C.: *The Game Loop*, In: Making Games. Apress, Berkeley, CA., https://doi.org/10.1007/978-1-4842-2493-9_2, 2016.

[7] Valente, Luis, Aura Conci, and Bruno Feijó: *Real time game loop models for single-player computer games*,Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment. Vol. 89. 2005.

[8] Zamith, Marcelo, Luis Valente, and Esteban Clua: *Game loop model properties and characteristics on multi-core cpu and gpu games*, SBGames 2016 (2016).

[9] *Gaffer On Games*, https://gafferongames.com/post/fix_your_timestep/, 2023.

[10] P. Mileff, J. Dudra: *Effective Pixel Rendering in Practice*, Production Systems and Information Engineering, Volume 10, No 1, pp. 1-15., 2022.

[11] *Game Programming Patterns - Game Loop*, https://gameprogrammingpatterns.com/game-loop.html, 2023.