The Game Loop and Real-Time Simulation

www.asyrani.com
Rendering Loop
In a graphical user interface (GUI), of the sort found on a Windows PC or a Macintosh, 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.
Older 2D video games used similar techniques to minimize the number of pixels that needed to be drawn.
Real-time 3D computer graphics are implemented in an entirely different way.

As the camera moves about in a 3D scene, the *entire contents* of the screen or window change continually, so the concept of invalid rectangles no longer applies.
The Game Loop

Run → Startup → Update → Stop → Shutdown

Wait → Draw

Do we have extra time?

No

Yes
Animation typically needs to be updated at a rate of 30 or 60 Hz, in synchronization with the rendering subsystem.

However, a dynamics simulation may actually require more frequent updates (e.g., 120 Hz).
Imagine that you took at most 6 images in one second.

Then, you try to create time lapse video that representing 6 images in 1 seconds.

However, you noticed that when you are playing your video, your video support up to 24 images per second

You are making a video that fast forward almost 6 times than the real life.
Camera (6 frames per second)

Projector (24 frames per second)

1 second
A Simple Example: Pong
In pong, a ball bounces back and forth between two movable vertical paddles and two fixed horizontal walls.

The human players control the positions of the paddles via control wheels. (Modern re implementations allow control via a joystick, the keyboard, or some other human interface device.)

If the ball passes by a paddle without striking it, the other team wins the point and the ball is reset for a new round of play.
void main() { // Pong
    initGame();
    while (true) // game loop {
        readHumanInterfaceDevices();
        if (quitButtonPressed()) {
            break; // exit the game loop
        }
        movePaddles();
        moveBall();
        collideAndBounceBall();
        if (ballImpactedSide(LEFT_PLAYER)) {
            incrementScore(RIGHT_PLAYER);
            resetBall();
        }
        else if (ballImpactedSide(RIGHT_PLAYER)) {
            incrementScore(LEFT_PLAYER);
            resetBall();
        }
        renderPlayfield();
    }
The original pong games were certainly not implemented by redrawing the entire screen at a rate of 30 frames per second.

Back then, CPUs were so slow that they could barely muster the power to draw two lines for the paddles and a box for the ball in real time.

Specialized 2D sprite hardware was often used to draw moving objects on-screen.
Game Loop Architecture
Technique 1: Windows Message Pumps
Technique 2: Callback-Driven Frameworks
Technique 3: Event-Based Updating

1. User interacts with page
   - Click me!

2. An "event" occurs

3. A piece of JS code runs in response
   - function myEvent() {
     ...
   }

4. The page's appearance is updated/modified in some way as a result
Abstract Timelines
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 (or some multiple thereof), although these time values can be easily converted into units of seconds by multiplying them by the frequency of the high-resolution timer on the current CPU.
Game Time

We can define a *game timeline* that is technically independent of real time. Under normal circumstances, game time coincides with real time.

If we wish to pause the game, we can simply stop updating the game timeline temporarily. If we want our game to go into slow-motion, we can update the game clock more slowly than the real-time clock. All sorts of effects can be achieved by scaling and warping one timeline relative to another.
Local and Global Timelines
Figure 7.1. Playing an animation clip can be envisioned as mapping its local timeline onto the global game timeline.
Figure 7.2. Animation play-back speed can be controlled by simply scaling the local time line prior to mapping it onto the global time line.
Figure 7.3. Playing an animation in reverse is like mapping the clip to the global time line with a time scale of $R = -1$. 
Measuring and Dealing with Time
Frame Rate and Time Deltas

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), which is the same thing as Hertz for all intents and purposes. Films traditionally run at 24 FPS.
**NTSC** delivers a frame rate of 30 fps at an aspect ratio of 720x480, and is used in North America, Japan and South Korea.

**PAL** is a different video standard that is incompatible with NTSC; it uses a fame rate of 25 fps and 720x576 aspect ratio, and is used in most of Europe, Australia and large parts of Africa and Asia.
The **differences between NTSC and PAL** are the reason why some DVDs or VHS tapes from Europe may not play in the United States and vice versa.

Most European DVD players can read NTSC and most PAL TVs can display NTSC video. But NTSC DVD players usually cannot read PAL.

There is a third standard, called SECAM (Sequential Couleur Avec Memoire or Sequential Color with Memory), that is used in Eastern Europe and France.
The duration between frames is often represented mathematically by the symbol $\Delta t$.

If a game is being rendered at exactly 30 FPS, then its delta time is $1/30$ of a second, or 33.3 ms (milliseconds).

At 60 FPS, the delta time is half as big, $1/60$ of a second or 16.6 ms.

$$\text{fps} = \frac{\text{number of frames}}{\text{second}}$$
We should note here that milliseconds are a common unit of time measurement in games. For example, we might say that animation is taking 4 ms, which implies that it occupies about 12% of the entire frame \((4 / 33.3 \approx 0.12)\).

Other common units include seconds and machine cycles.
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 (or in a 2D game, we might specify this as 40 pixels per second!)
One simple way to accomplish this is to multiply the ship’s speed $v$ (measured in meters per second) by the duration of one frame $\Delta t$ (measured in seconds),

yielding a change in position $\Delta x = v \Delta t$ (which is measured in meters per frame). This position delta can then be added to the ship’s current position $x_1$, in order to find its position next frame: $x_2 = x_1 + \Delta x = x_1 + v \Delta t$. 
Dealing with Time in Developing Game Engine and Game

- Measuring Real Time with a High-Resolution Timer
- Time Units and Clock Variables
- Dealing with Break Points
- A Simple Clock Class
Read Related Topics
Try some game engine with code
Goodluck in coding