Javascript Game Development - The Game Loop

Arthur Schreiber · 2 February 2011

One of the most important parts of a game engine is the so called "game loop". It is the central piece of the game's engine and is responsible for trying to balance running a game's logic, and executing its drawing operations.

A very basic game loop would look something like this in JavaScript (this does not work in Browsers!):

```
var Game = { };

Game.draw = function() { ... draw entities here ... };
Game.update = function() { ... run game logic here ... };

while (!Game.stopped) { // While the game is running
  Game.update();        // Update Entities (e.g. Position)
  Game.draw();          // Draw Entities to the Screen
}
```

Writing a game loop for execution in the browser is a tad more tricky than that. We can't just use a while loop that runs forever, as JavaScript execution blocks the browser from repainting its window, making the game seem to have "locked-up".

So we'll have to try "emulating" a real loop, while giving back control to the browser after every drawing operation to not lock-up the interface.

setInterval to the rescue!

window.setInterval is exactly what we're looking for. It provides a way to run code in a loop while allowing the browser to repaint the window in between the loop runs.

```
Game.fps = 50;

Game.run = function() {
  Game.update();
  Game.draw();
};

// Start the game loop
Game._intervalId = setInterval(Game.run, 1000 / Game.fps);
```

```
...

// To stop the game, use the following:
clearInterval(Game._intervalId);
```

Check out the example page for a game loop that uses setInterval.

That's it?

Well, yeah. At least for a basic game. The main problem with this kind of loop is that the updating and drawing operations are "glued" together (this means your game's logic will run as fast as your drawing operations). So if your frame rate drops below 60fps, your game will also seem to be running slower.

For some games, this might never be a problem, for others this behaviour can lead to jerkiness. Also, if you want to add a multiplayer component, you want to make sure that your logic runs with the same speed on all your clients.

This can be achieved using a so called "game loop with fixed time steps". Basically, we try to run the logic a fixed amount of times per second, while trying to draw as many frames per second as possible.

```
Game.run = (function() {
  var loops = 0, skipTicks = 1000 / Game.fps,
      maxFrameSkip = 10,
      nextGameTick = (new Date).getTime();

  return function {
    loops = 0;

    while ((new Date).getTime() > nextGameTick && loops < maxFrameSkip) {
      Game.update();
      nextGameTick += skipTicks;
      loops++;
    }

    Game.draw();
  };
})();

// Start the game loop
Game._intervalId = setInterval(Game.run, 0);
```

Now, this will give us a game loop that runs 50 Game updates per second, while trying to run the drawing code as often as possible.

This gives very, very smooth animations, on my machine, I get around 200 drawing operations per second with Google Chrome, while the number of updating operations stays constant at 50 per second.

Check out the example.

More problems?!

But now we’re facing another problem: We’re burning a lot of CPU cycles, way more than are actually needed to give our players smooth animations. Most computer screens have a refresh rate of either 50 or 60 Hz, so having 200 rendering operations per second is total overkill, and brings my CPU usage up to 48%, according to Chrome’s Task Manager.

That’s why Mozilla introduced window.mozRequestAnimationFrame (which also has a WebKit twin called window.webkitRequestAnimationFrame) a way to easily limit the number of drawing operations per second to the number of screen refreshes per second. (Currently, both mozRequestAnimationFrame as well as webkitRequestAnimationFrame do not truly sync with the screen refresh rates, but simply help in limiting drawing operations to a sane amount, but real sync will be implemented eventually.)

To be also compatible with browsers that do not implement window.*RequestAnimationFrame, we’ll have to provide a fallback based on setInterval, but limited to 60 frames per second:

```
(function() {
  var onEachFrame;
  if (window.webkitRequestAnimationFrame) {
    onEachFrame = function(cb) {
      var _cb = function() { cb(); webkitRequestAnimationFrame(_cb); }
      _cb();
    };
  } else if (window.mozRequestAnimationFrame) {
    onEachFrame = function(cb) {
      var _cb = function() { cb(); mozRequestAnimationFrame(_cb); }
      _cb();
    };
  } else {
    onEachFrame = function(cb) {
      setInterval(cb, 1000 / 60);
    }
  }

  window.onEachFrame = onEachFrame;
})();
```

```
window.onEachFrame(Game.run);
```

This gives us smooth game play without totally burning the CPU (according to Chrome's Task Manager, this gives a CPU load of 10-20% when using webkitRequestAnimationFrame).

See for yourself here.

Conclusion

Wow, that's been a long journey we had to undertake to get a game loop that's giving us both smooth animations while making sure that our game runs at a constant pace.

I'll try to post some more articles about game development using Javascript and HTML5 over the next weeks, so make sure to check back from time to time!

Edit - February 3rd, 2010

As was pointed out in the comments, there is no benefit in running more drawing operations than update operations (except for maybe benchmarking), so you have to either interpolate or skip drawing if there were absolutely no game updates.

Skipping drawing operations is actually really easy to achieve, just put a small condition around the call to Game.draw:

```
if (loops) Game.draw();
```

This makes sure that we only draw the game if we have run at least one updating operation.

Interpolation is a bit more difficult, as you have to make all drawing operations aware of the interpolation, and it will make your game drawing code considerably more complicated.

Here is the code for an interpolating game loop:

```
// Updated drawing code for our objects
Rect.prototype.draw = function(context, interpolation) {
  context.fillRect(this.x, this.y + this.velocity * interpolation, 30, 30);
};
```

```
Game.draw = function(interpolation) {
  this.context.clearRect(0, 0, 640, 480);

  for (var i=0; i < this.entities.length; i++) {
    this.entities[i].draw(this.context, interpolation);
  }
};

Game.run = (function() {
  var loops = 0, skipTicks = 1000 / Game.fps,
      maxFrameSkip = 10,
      nextGameTick = (new Date).getTime(),
      lastGameTick;

  return function() {
    loops = 0;

    while ((new Date).getTime() > nextGameTick) {
      Game.update();
      nextGameTick += skipTicks;
      loops++;
    }

    if (!loops) {
      Game.draw((nextGameTick - (new Date).getTime()) / skipTicks);
    } else {
      Game.draw(0);
    }
  };
})();
```

Now, if you're going to do interpolation, you should also lower the number of game updates you run, down to maybe 25 or 30 updates per second (but don't forget to update the movement speed of your objects accordingly).

29 Comments NoKarma.org Login

Sort by Best Share Favorite

Join the discussion…

louisremi • 4 years ago

In your second example, I don't understand why you want draw more often than you run you're game logic...
If the 'state' of the game hasn't updated, why would you redraw the scene?
Last but not least, nothing tells you that Chrome actually performs a 'repaint' as often as your Game.draw() function runs. 200 clearly seems to be an unrealistic repaint rate to me.
Firefox introduced a window.mozPaintCount to offer a way to count those paints, and there is an open bug for webkit http://code.google.com/p/chrom...
Great article, btw.

6 ^ | ˅ • Reply • Share ›

hello ➦ louisremi • a month ago

Interpolation for visual appeal.

^ | ˅ • Reply • Share ›

Kubica • a year ago

Problem with this is that, your physics loop will pause when you change browser tab.

2 ^ | ˅ • Reply • Share ›

shan • 2 years ago

suppereb

1 ^ | ˅ • Reply • Share ›

louisstow • 4 years ago

Great tutorial. How would you calculate the FPS with your implementation?

1 ^ | ˅ • Reply • Share ›

Arthur Schreiber Mod ➦ louisstow • 4 years ago

In the examples I used Mr.doob's Stats.js, which is pretty nice. See http://mrdoob.com/blog/post/70... and https://github.com/mrdoob/stat.... Take also a look at the examples' source.

^ | ˅ • Reply • Share ›

louisremi • 4 years ago

And how can you do interpolation when you don't know what the next state of the game can be?
To me, the only way to run a game at a constant speed is to calculate the time elapsed since the last game update and animate the scene proportionally.
For example, if you have to move a sprite on a scene: you would move it by 1 pixel if last game update was 13ms ago; but you would move it by two pixels if last game update was 26.
That's the way animation work in jQuery, for sure.

You can even use jQuery animate function as an animation loop, see
http://jsfiddle.net/louisremi/...

1 ^ | ˅ • Reply • Share ›

kaoD ➦ louisremi • 3 years ago

You can always interpolate with the last frame, not the next. You're effectively one frame behind, but it's the only way around and not really noticeable.

^ | ˅ • Reply • Share ›

Arthur Schreiber Mod ➦ louisremi • 4 years ago

You could take the difference between (new Date().getTime()) and nextGameTick right before drawing and use that as an interpolation value.

^ | ˅ • Reply • Share ›

DR. MOO • a year ago

MOOOOOOOOOOOOOOOO. : D <3

1 • Reply • Share ›

Salva • 3 months ago

Curiously, I developed a JS library for simulation loops. Take a look at: https://github.com/lodr/strong...

I based it on http://gafferongames.com/game-... article.

• Reply • Share ›

yanko • a year ago

are you planing to continue this?

• Reply • Share ›

Arthur Schreiber Mod yanko • a year ago

Not really, no. I've not been having any free time to look into this more closely and/or write blog posts on related topics.

• Reply • Share ›

Iain • 4 years ago

Just a minor point, from experience your better using setTimeout rarther than setInterval, as setInterval will try and call your main game loop after a fixed period of time. The issue with this is that if your game loop hasn't finished processing your pushing more and more onto the stack, which over time causes high memory usage, and browser crashes.

You best bet is to use a recursively called game loop that uses setTimeout. What setTimeout does is waits a period of time before executing the next operation, this will prevent the stack getting full, and lessen the risk of a browser crash.

See http://pastebin.com/KGZ2wYBP for an example of the above.

• Reply • Share ›

Arthur Schreiber Mod Iain • 4 years ago

I did not mention setTimeout on purpose. :) The problem with setTimeout is that most browsers have a minimum timeout value you can pass, if you pass it a smaller value, it will use the built-in minimum. (IIRC, for Chrome that is 1ms, FF and IE around 15, no idea about Opera).

Now, if the function you pass to setTimeout takes 10ms, and the pause between each setTimeout takes 16, your passed function will only be run every 26ms in effect. If you use setInterval, your function will be run every 16 ms.

2 • Reply • Share ›

Iain Arthur Schreiber • 4 years ago

Ah sorry to mention it :)

I wasn't aware of the minimum timeout value. A while back I'd wrote some fps monitoring code for a silly little game project I was working on. The fps code would call 'new Date().getTime()' to get the time is MS since the epoch and then minus the prior frames timestamp giving me my fps.

With nothing in the game loop apart from the fps code I was hitting 1000 fps in FF and Chrome, which is the logical max for a setTimeout(func, 1); loop. I'll have to go back and check that my code was working as intended, and that my memory isn't playing tricks on me.

2 • Reply • Share ›

Arthur Schreiber Mod Iain • 4 years ago

See
http://ajaxian.com/archives/se...
https://developer.mozilla.org/...
http://code.google.com/p/chrom...

So your result should be totally correct for Chrome as it is having a clamp of 1ms, but you might get different results for different versions of Firefox.

3 • Reply • Share ›

Iain Arthur Schreiber • 4 years ago

Ah thanks, looks like I need to serve myself a slice of fail pie :)

4 • Reply • Share ›

knights_who_say_ni Iain • 7 months ago

We need "experts" like you to chime in on every thread.

• Reply • Share ›

louisremi • 4 years ago

The biggest misconception here is that you assume that you will be able to draw the scene more often than the game logic runs. This is wrong in most case: drawing the scene is the most expensive task your game engine will have to perform.

• Reply • Share ›

kaoD louisremi • 3 years ago

Nope. The biggest misconception here is that drawing more often than updating will lead to smoother animations. If you draw something that hasn't changed, you're just wasting power.

• Reply • Share ›

Arthur Schreiber Mod louisremi • 4 years ago

Yes, exactly. As soon as you start drawing more complex things, like images, your framerate will drop significantly. The idea is to keep the number of game updates constant, so it makes no difference to the way things behave in your game due to the players hardware or browser :) if drawing gets too complex,

Reply • Share ›

dom • 4 years ago

There are two basic approaches for game loops: fixed timestep with frameskip (skip drawing for 1 or more frames) or a variable timestep.

I think you got the first approach the wrong way around :)

Usually drawing a frame is far more expensive than updating the game state. So if your computer is too slow to render 30fps it's because it can't draw fast enough. Now, to not slow the game down or introduce any quirks, you usually skip the drawing for a frame or two and just do the update. I.e.: frameskip.

The other thing you can do is to measure the time since the last frame and use this "tick" value in all your movement and physic calculations.

Typically the first approach is used by emulators or older games, while all (to my knowledge) "modern" games use a variable timestep. The FlashPunk engine is a notable exceptions - it can do both.

Reply • Share ›

Arthur Schreiber Mod dom • 4 years ago

I don't think so. If you look more closely at the example code, you'll see that there is a loop that runs until the number of updating operations matches the number of updates that should have been run This, in turn, will skip drawing of frames.

The second loop condition limits the number of skipped frames, so the players can see some game redraws from time to time.

[Edit]
Microsoft's XNA Framework also allows you to choose between fixed- and variable-step game loops. It's just that I think that fixed game steps are a lot easier to handle. :)

Reply • Share ›

dom Arthur Schreiber • 4 years ago

Oh, okay. Sorry. I misinterpreted this paragraph, thinking that you always skip updates in favor of drawing:

"(...) I get around 200 drawing operations per second with Google Chrome, while the number of updating operations stays constant at 50 per second."

So you're actually doing both, right? Skipping updates when you already have 50 per second and skip drawing when you have less than 50 updates per second.

As you pointed out in the comments, drawing more often than updating only makes sense when you interpolate movement in between updates. But you'd have to predict (interpolate) movement into the future for that!? Doesn't this introduce more "artifacts"?

I'm still not convinced that drawing more often than updating is a good idea :)

Reply • Share ›

Arthur Schreiber Mod → dom • 4 years ago

Yes, and it probably is not. :) But it's a nice way to measure the performance of your drawing code.

I just updated the article with an example on how to either limit the drawing operations or do interpolation.

Reply • Share ›

Johan • 4 years ago

The reason you need more draw frames than logic update frames is to provide the opportunity to do smooth animation (possibly with interpolation). At 60fps logic updates it might not make a difference but at 30fps logic and 60fps draw with interpolation you can make your game visually smoother, but save the logic update cycles if you don't need them.

Reply • Share ›

ajuc → Johan • 3 years ago

Also multiplayer code is much easier if there is fixed update framerate. But draw framerate can be different for each player, to allow slower computers to run the game. And players with better computers gets smother graphics because of interpolation.

Reply • Share ›

Stephane Roucheray • 4 years ago

In your second example, to drop the number of updates and draws you could use invalidation as well :

```
Game.run = (function() {
var loops = 0, skipTicks = 1000 / Game.fps,
maxFrameSkip = 10,
nextGameTick = (new Date).getTime();

return function {
loops = 0;

while ((new Date).getTime() > nextGameTick && loops < maxFrameSkip) {
if(Game.updateInvalidation){
Game.update();
Game.updateInvalidation = false
}
nextGameTick += skipTicks;
loops++;
}

Game.draw();
};
})();
```

The Game.updateInvalidation should be set to true whenever, in the game, something modifies the game environment that will require an update. Thus you update the game if, and only if, an invalidation is asked.