Contents:

# 1 Introduction

# 2 Pacing

2.1 Why frame rates vary

2.2 Callbacks

2.3 Using elapsed_ticks

2.4 Recipes

2.5 Basic Motion

2.6 Motion with acceleration

2.7 Motion with friction

2.8 Game timers

2.9 Validating your implementation

# 3 Indices and tables
A Javascript vector graphics game framework
Flynn is a vector graphics gaming framework for Javascript. It is the engine behind all the games posted at VectorAlchemy.

I wrote Flynn to support my own game development. Along the way I’ve worked aggressively to push as much generalizable functionality into it as possible. Games implemented on top of Flynn end up having a light code base, and the engine handles all the pesky browser issues (touch events, window resizing, etc.). In practice, new game development stays focused on game mechanics.

Games written on top of Flynn can run cross-platform (desktop/mobile) “out of the box” with little effort. If you’re mildly enterprising then the Flynn test app code and the code for the existing VectorAlchemy games should be sufficient for you to develop your own games using Flynn.

I’m just starting to author this documentation so it’s pretty light right now. My plan is to hit the most non-intuitive issues first, then we’ll see what happens next.

My minimum goal is to document:

• Pacing (the use of paceFactor in callbacks)
• Developer mode
• Polygon editing

Welcome aboard!
“Pacing” is used to control the perceived speed a game runs at. Flynn strives to render games at 60 frames per second (FPS), but several factors can reduce the actual frame rate achieved by a game. A properly written game should run at a consistent speed regardless of the frame rate it achieves from moment to moment.

For example, suppose you write a game in which an airplane drops a bomb. Running on your state-of-the-art laptop the game renders at 60FPS and the bomb takes 1 second to fall from the top of the screen to the bottom. Now you run the same code on an old PC and, because the processor is slower, it only renders at 30FPS. The animation will look choppier on the slower machine, but when you drop a bomb it should still take 1 second to fall. Flynn’s pacing infrastructure will help you achieve that.

### 2.1 Why frame rates vary

The frame rate of a game is primarily affected by four things:

1. **CPU performance**: Slower CPUs may render at a lower frame rate than faster CPUs.
2. **CPU Load**: Other processes running on the machine may overburden the CPU, causing the frame rate to drop.
3. **Rendering Time**: At 60FPS, a game has 1/60 of a second (.0167 seconds) to update the game world and draw it. For a game to render at 60FPS, the combined time for the browser to process an animation frame, the game to update the state of the game world, and the game to draw the new frame, must not exceed 0.167 seconds. If it does, the frame rate will drop.
4. **Update Time**: See previous.

### 2.2 Callbacks

The Flynn engine makes 3 callbacks to your game code each animation frame. Nominally (if maintaining 60FPS), they each get called 60 times per second:

- **handleInputs(input, elapsed_ticks)**: Processes all user input to your game (button presses, screen touch events).
• `update(elapsed_ticks)`: Updates the state of the game world.
• `render(ctx)`: Draws the game world

### 2.3 Using `elapsed_ticks`

Flynn calls each of your 3 callback functions once per frame. Each time Flynn calls `update()` and `handleInputs()` your game will change the game world a little. In practice, you need to know how much time has elapsed since the last call so that you can change the game world by the appropriate amount. If 16.6 milliseconds have elapsed (1/60 of a second), then you may, for example, need to move a particular space ship 1 pixel. But if 33.3 milliseconds have elapsed (1/30 of a second), then you should move that ship 2 pixels.

Flynn tells you how much time has elapsed by passing you a the parameter `elapsed_ticks`, which tells you how many nominal 60FPS game “ticks” have elapsed. If your game is running at 30FPS, then Flynn will pass a `elapsed_ticks` of 2.0 to your callbacks.

If you’d like to know the elapsed time in seconds, you can always calculate it like this:

```javascript
update: function(elapsed_ticks) {
    // ...
    var elapsed_seconds = elapsed_ticks / Flynn.TICKS_PER_SECOND;
    // Note: Flynn.TICKS_PER_SECOND always == 60
    // ...
}
```

In general, you’ll use `elapsed_ticks` to calculate the appropriate movement of objects by doing something like:

```javascript
position = position + velocity * elapsed_ticks;
```

Below you’ll find code recipes for various common pacing scenarios.

### 2.4 Recipes

Here are some code recipes for using `elapsed_ticks` to update positions, velocities, and timers.

In the vectorized versions below, variables with the `_v` suffix are 2d vectors. They are instances of type `Victor`, implemented by the `victorjs` library.

In all recipes below, variables have the following units:

- Position: pixels
- Velocity: pixels/tick (i.e. pixels per tick)
- Acceleration: pixels/tick^2 (i.e. pixels per tick per tick)
- Time Constant: ticks

**Note:** The Flynn demo application implements the vector form of the recipes below (see the “PACING” tab in the demo), and shows their behavior at 60FPS and at 20FPS to validate that they behave the same way in both cases.
2.5 Basic Motion

Governing equations:

• position_new = position_old + velocity * elapsed_time

```plaintext
//--------------
// Single Axis form
//--------------

// Position
position_y += velocity_y * elapsed_ticks;

//--------------
// Vectorized form
//--------------

// Position
position_v.add(velocity_v.clone().multiplyScalar(elapsed_ticks));
```

2.6 Motion with acceleration

Governing equations:

• position_new = position_old + acceleration * elapsed_time^2 / 2 + velocity_old * elapsed_time
• velocity_new = velocity_old + acceleration * elapsed_time

```plaintext
// Where acceleration is in units of pixels/tick^2 (i.e pixels per tick per tick)

//--------------
// Single Axis form
//--------------

// Position (from equivalent physics equation p1 = P0 + a*t^2/2 + v0*t)
position_y += acceleration_y * Math.pow(elapsed_ticks, 2) / 2 + velocity_y * elapsed_ticks;

// Velocity
velocity_y += acceleration_y * elapsed_ticks;

//--------------
// Vectorized form
//--------------

// Position (from equivalent physics equation p1 = P0 + a*t^2/2 + v0*t)
position_v.add(acceleration_v.clone().multiplyScalar(Math.pow(elapsed_ticks, 2) / 2).add(velocity_v.clone().multiplyScalar(elapsed_ticks)));
```

(continues on next page)
// Acceleration
velocity_v.add(
    acceleration_v.clone().multiplyScalar(elapsed_ticks)
);

2.7 Motion with friction

Friction is modeled using an exponential decay time constant. The time constant is equal to the time, in game ticks, for the velocity to decay to about 37% of its initial value (1/e \( 0.36788 \)).

Governing equations:

- \( \text{position}_\text{new} = \text{position}_\text{old} + (\text{velocity}_\text{old} - \text{velocity}_\text{old} \times e^{(-\text{elapsed}_\text{time} / \text{friction}_\text{time}_\text{constant})} \times \text{friction}_\text{time}_\text{constant} \)
- \( \text{velocity}_\text{new} = \text{velocity}_\text{old} \times e^{(-\text{elapsed}_\text{time} / \text{friction}_\text{time}_\text{constant})} \)

//---------------
// Single Axis form
//---------------
// Note: The position must be updated before the velocity, since the position
equation presumes velocity_y is the old velocity.

// Position
position_y +=
    (velocity_y - velocity_y \times Math.pow(Math.E, -elapsed_ticks / friction_time_constant)) \times
    friction_time_constant;

// Velocity
velocity_y *= Math.pow(Math.E, -elapsed_ticks / friction_time_constant);

//---------------
// Vectorized form
//---------------
// Note: The position must be updated before the velocity, since the position
equation presumes velocity_v is the old velocity.

// Position
position_v.add(
    velocity_v.clone().subtract(
        velocity_v.clone().multiplyScalar(
            Math.pow(Math.E, -elapsed_ticks / friction_time_constant)
        ).multiplyScalar(friction_time_constant)
    ).multiplyScalar(friction_time_constant)
);

// Velocity
velocity_v.multiplyScalar(Math.pow(Math.E, -elapsed_ticks / friction_time_constant));
2.8 Game timers

To keep track of the total elapsed time (in seconds) in your game, you need to accumulate the elapsed time for each game tick.

```javascript
update: function(elapsed_ticks) {
    // ...
    // Game timer in seconds
    this.game_timer_sec += elapsed_ticks / Flynn.TICKS_PER_SECOND;
    // Game timer in ticks
    this.game_timer_ticks += elapsed_ticks;
    // ...
}
```

Note: `Flynn.mcp.gameSpeedFactor` is a single parameter which can be used to speed up or slow down an entire game. It gets passed as a parameter to `Flynn.Mcp.init()` during initialization to set the overall speed of a game. Setting the `gameSpeedFactor` to a value other than 1.0 causes Flynn to lie to you when it calls `handleInputs()` and `update()`, by artificially scaling `elapsed_ticks` accordingly.

2.9 Validating your implementation

Properly applying `elapsed_ticks` in your game logic can be tricky. Personally I get it wrong *all the time*. But don’t worry, Flynn has your back!

First, activate “Developer mode”, by adding “?develop” to the end of your game’s URL in the browser (for example, you can start Roundabout in developer mode like this:


There are three “Developer mode” features specifically designed to help you validate pacing…

- **FPS20**: Press the \ key to toggle FPS20 mode. The text “FPS_20” will appear in the lower-left corner of the screen, and your game will be forced to run at 20 frames per second. If you’ve applied `elapsed_ticks` correctly, the “speed” of your game should remain unchanged. If, for example, you discover that in FPS20 mode your space ship moves at normal speed but its bullets move slowly, then you have forgotten to apply `elapsed_ticks` to your bullet code.

- **SLOWMO**: Press the 7 key to toggle SLOWMO mode. The text “SLOWMO” will appear in the lower-left corner of the screen, and your game will be forced to run in slow motion (though still at nominally 60FPS). If you’ve applied `elapsed_ticks` correctly, everything should be super slow. If, for example, you discover that your space ship moves slowly in SLOWMO mode but its bullets move at their normal speed, then you have forgotten to apply `elapsed_ticks` to your bullet code.

- **STATS**: Press the 6 key to toggle the STATS display. Flynn will show you four real-time graphs:
  - **Yellow**: The actual frame rate of your game.
  - **Dark Blue**: The total (combined) time (in milliseconds) spent in your `handleInputs()`, `update()`, and `render()` methods.
– **Light Blue**: The total (combined) time (in milliseconds) spent by the WebGL renderer (PixiJS) to draw frames.

– **Magenta**: The total (combined) time spent in the browser’s animation callback (includes game logic, Pixi rendering and browser rendering). If this time exceeds 16.6ms, your frame rate will drop below 60FPS.

**Note:** The keyboard keys assigned to the various “Developer Mode” functions can vary from game to game. Most “Official” Flynn games bind them to the values shown above, but you can bind them to anything (or nothing). When in doubt, you can find a game’s key bindings in its `main.js` file.
CHAPTER 3

Indices and tables

• genindex
• modindex
• search