WebGL Animation

Animation vs Simulation

Suppose we want to show a bouncing ball – we can try to model it accurately using Physics or we can animate using some other control mechanism.

To illustrate this we'll animate a bouncing ball.

Animation

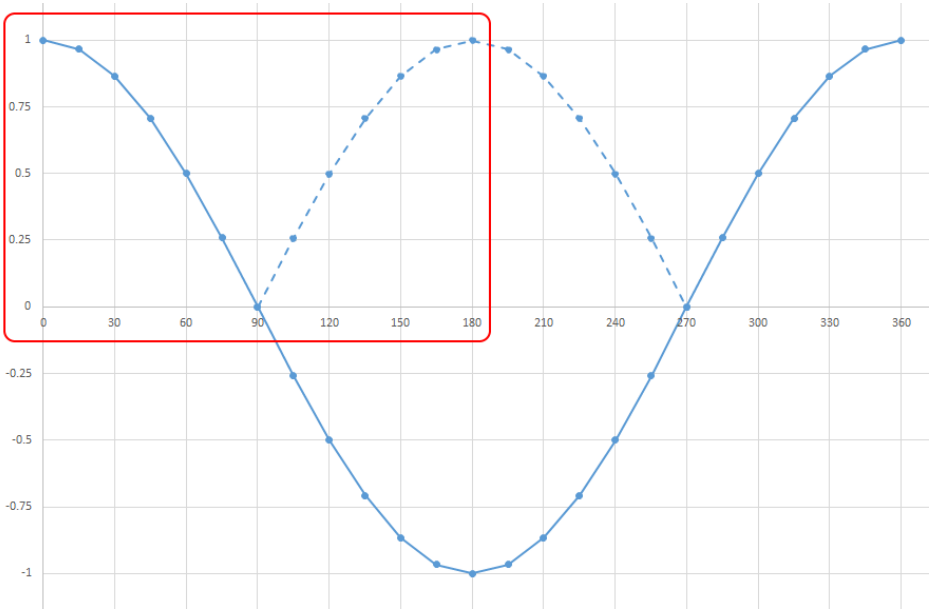
This time we'll be using Babylon.js, below is the base script.

```
<!DOCTYPE html>
<html>
<head>
  <title>Animation vs simulation</title>
  <meta charset="utf-8" />
  <script src="babylon.js"></script>
  <style>
   html, body {
      overflow: hidden;
      width: 100%;
      height: 100%;
      margin: 0;
      padding: 0;
    }

    #renderCanvas {
      width: 100%;
      height: 100%;
      touch-action: none;
    }
  </style>
</head>
<body>
  <canvas id="renderCanvas"></canvas>
  <script>
    window.addEventListener('DOMContentLoaded', function () {
      var canvas = document.getElementById('renderCanvas');
      var engine = new BABYLON.Engine(canvas, true);
      var sphere0;
      var sphere1;

      var time = Date.now();

      var createScene = function () {
        var scene = new BABYLON.Scene(engine);
        var camera = new BABYLON.ArcRotateCamera("ArcRotateCamera", 0, 0, 0, new BABYLON.Vector3(20, 10,
0), scene);
        camera.setTarget(BABYLON.Vector3.Zero());
        camera.target.y = 5;
        camera.attachControl(canvas, false);

        var light = new BABYLON.HemisphericLight('light1', new BABYLON.Vector3(0, 1, 0), scene);

        sphere0 = BABYLON.Mesh.CreateSphere('sphere0', 16, 2, scene);
        sphere0.position.y = 10;
        sphere0.position.z = 1.5;

        sphere1 = BABYLON.Mesh.CreateSphere('sphere1', 16, 2, scene);
        sphere1.position.y = 10;
        sphere1.position.z = -1.5;

        var ground = BABYLON.Mesh.CreateGround('ground1', 6, 10, 2, scene);
        ground.position.y = -1;

        return scene;
      }
      var scene = createScene();
```

```
      // animate balls
      // 1 via "physics"
      var Physics = {
        s: 10,
        update: function (t) {
        }
      }

      // 2 via tweening (use cos/sin)
      var Tween = {
        update: function (t) {
        }
      }

      var then = time, duration = 0;
      engine.runRenderLoop(function () {
        duration = time - then;
        then = time;
        time = Date.now();

        Physics.update(duration/1000);
        sphere0.position.y = Physics.s;

        Tween.update(duration / 155)
        sphere1.position.y = Tween.s * 10;

        scene.render();
      });

      window.addEventListener('resize', function () {
        engine.resize();
      });
    });
  </script>
</body>
</html>
```

Animation

A sine wave follows a similar path, slowing at the top and bottom of the path, while accelerating in-between.



However, we need the ball to bounce -- stopping abruptly and reversing its direction. This can be done by negating (× -1) the cosine for values between 90° and 270° and, since cosine values greater than 270° repeat, we can limit the range simply by resetting the value.

Finally, a sine function returns values between 1 and -1, so we just need to scale it up to the desired range.

Also, we need to pass t to the update function to ensure the animations advance by the correct each time.

Usually, for animation there is some limiting factor (usually time). Here, for simplicity's sake (the animation repeats indefinitely) I haven't implemented such a factor.

```
// 2 via tweening (use cos)
var Tween = {
  a: 0,
  s: 0,
  update: function (t) {
    this.a += t / (2 * Math.PI);
    if (this.a > Math.PI)
      this.a -= Math.PI

    if (this.a > Math.PI / 2)
      this.s = -Math.cos(this.a);
    else
      this.s = Math.cos(this.a);
  }
}
```

Simulation

For this we need some basic physics and maths.

A ball will fall under the force of gravity with a constant acceleration of 9·8m/s².

So at time, t, its velocity, v, is given by:

v = u + at (where u is its initial velocity.

Its displacement (or position), s, is given by:

s = ((u + v) / 2) × t

Again, we need to pass t to the update function to ensure the animations advance by the correct each time.

```
var Physics = {
  a: -9.8,
  v: 0,
  s: 10,
  update: function (t) {
    if (t > 0.02) t = 0.02;

    this.v = this.v + this.a * t;
    this.s += this.v * t;
    if (this.s < 0) {
      this.s = 0;
      this.v = -this.v * 1.007;
    } else {
      if (this.s > 10) this.s = 10;
    }
  }
}
```

Summary

Both approaches can be prone to errors, the physics based approach requires accurate maths and collision detection, and appropriate time steps. The animation approach is, by its nature, an approximation at best.

So, it's best to use an existing library unless you're mathematically inclined and wish to spend hours developing and debugging your own.

requestAnimationFrame()

Here we're using Babylon.js's `runRenderLoop` which almost certainly uses `requestAnimationFrame` behind the scenes. For our own animations we should also use `requestAnimationFrame`.

There may be a temptation to use a timer function (`setTimeout()` or `setInterval()`) but there are problems with doing so:

- The timer functions call callbacks at a specific interval (or as close to it as possible), regardless of whether it is a good time to draw or not. †
- JavaScript executed in a timer callback has no reliable way to synchronize with the timing of other browser-generated animation on the page (e.g., SVG or CSS Transitions). †

- Timers execute regardless of whether a page or tab is visible or the browser window has been minimized, potentially resulting in wasted drawing calls. †
- JavaScript application code has no idea of the display's refresh rate and so has to make an arbitrary choice for the interval value: make it 1/24 of a second, and you deprive the user of resolution on a 60 Hz display; make it 1/60 of a second and on slow-refresh displays, you waste CPU cycles drawing content that is never seen. †

`requestAnimationFrame()` was designed to solve all of the preceding problems. †

† Taken from **Programming 3D Applications with HTML5 and** WebGL by *Tony Parisi*

`requestAnimationFrame()` calls a given function before the next repaint, providing a reliable method of synchronising animations on the web page.

The `requestAnimationFrame()` callback function is only called when the browser page is visible, which mean less CPU, GPU, and memory usage, leading to longer battery life.

Conversely, if you want something to happen whether or not the browser is visible or active **don't use** `requestAnimationFrame`.

DIY

The previous two implementations are a bit noddy but show the basic requirements for an animation function, a control step usually based on time, an output value and control logic.

I developed a WebGL "widget" for a Khronos a couple of years ago.

Most of the processing was done on the GPU, the shaders are quite long, with mainly control being done in JavaScript.

The code sprawls a bit, it was written in a hurry, but the animating functions share the same qualities plus a limiting factor. Here, it's a different factor for each animation function: angle, position, transparency, etc..

You can also see some repetition, a more generic function could have been developed and used.

Again, a very good reason to use a third party library.

Tween.js

Tween.js is a library developed by Solded Penadés (github.com/tweenjs/tween.js/) and is used in a number of projects including A-Frame (a-frame.io) and Rome "3 Dreams of Black" (ro.me).

(There's another Tween.js which is part of the CreateJS suite (www.createjs.com). There are likely others.)

https://tweenjs.github.io/tween.js/examples/03_graphs.html

Obviously, not limited changing on object's position, any property is fair game.

Even the material can change of time.

KeyFrame.js

Data structures represent individual values along a timeline, and an engine calculates (interpolates) intermediate values to produce a smooth result. Unlike tweens, which support a single transition from one value to another, key frames allow us to create a series of transitions within one animation.

Some Other Methods

1. Animation along a path.
2. Morphing.
3. Skinning. - https://threejs.org/examples/webgl_animation_skinning_morph
4. Rigging.

Physics engines

http://alteredqualia.com/xg/examples/animation_physics_level.html