# The performance of this game is very machine/browser dependent. It works quite well in modern browsers, especially those with GPU canvas acceleration, but a bad graphics driver can kill it stone dead. So your mileage may vary. There are controls provided to change the rendering resolution and the draw distance to scale to fit your machine.

# Currently supported browsers include:

# Firefox (v12+) works great, 60fps at high res - Nice!

# Chrome (v19+) works great, 60fps at high res... provided you dont have a bad GPU driver

# IE9 - ok, 30fps at medium res... not great, but at least it works

# The current state of mobile browser performance is pretty dismal. Dont expect this to be playable on any mobile device.

# *NOTE: I havent actually spent anytime optimizing for performance yet. So it might be possible to make it play well on older browsers, but that's not really what this project is about.*

# **A note on code structure**

# This project happens to be implemented in javascript (because its easy for prototyping) but is not intended to demonstrate javascript techniques or best practices. In fact, in order to keep it simple to understand it embeds the javascript for each example directly in the HTML page (horror!) and, even worse, uses global variables and functions (OMG!).

# If I was building a real game I would have much more structure and organization to the code, but since its just a racing game tech demo, I have elected to [KISS](http://en.wikipedia.org/wiki/KISS_principle).

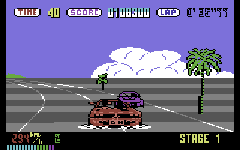
# 

# 

# Build a racing game

I wasn’t really much for going to the arcades when I was younger… I didn’t need ‘em with awesome C64 games sitting at home… but there were 3 arcade games that could always get my money - Donkey Kong, Dragons Lair, and Outrun…

… and I really loved Outrun, the speed, the hills, the palm trees and the music - even the lowly c64 version.



So, I wanted to try my hand at an old-school pseudo-3d racing game, a-la outrun, pitstop or pole-position. I dont plan on building a fully fleshed out and finished game but thought it would be fun to re-examine the mechanics of how these games pulled off their tricks. The curves, the hills, the sprites and the feeling of speed…

# 

# 

# 

# 

# 

# 

# Straight roads

Well, we’re going to need to

* revise some trigonometry
* revise basic 3d projection
* build a game loop
* load some sprite images
* build some road geometry
* render the background
* render the road
* render the car
* enable keyboard support to drive the car

Before we do any of that, lets go off and read [Lou’s Pseudo 3d Page](http://www.extentofthejam.com/pseudo/) - its the main source of information (that I could find) online about how to build a pseudo-3d racing game.

*NOTE: Lou’s page doesn’t render well in google chrome - so its best viewed using Firefox or IE*

Finished reading Lou’s article ? Great! We’re going to build a variation on his ‘Realistic Hills Using 3d-Projected Segments’ approach. We will do it gradually, over the course of the next 4 articles, but we will start off here with v1, building very simple straight road geometry and projecting it onto our HTML5 canvas element.

see it in action here

## Some Trigonometry

Before we get down to the implementation, lets use some basic trigonometry to remind ourselves how to project a point in a 3D world onto a 2D screen.

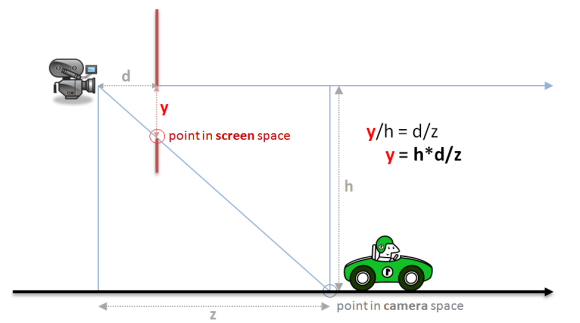
At its most basic, without getting into vectors and matrices, 3D projection uses a law of [similar triangles](http://www.khanacademy.org/math/geometry/triangles/v/similar-triangles). If we were to label:

* h = camera height
* d = distance from camera to screen
* z = distance from camera to car
* y = screen y coordinate

Then we could use the law of similar triangles to calculate

y = h\*d/z

as shown in the diagram below:



We could have also drawn a similar diagram from a top-down view instead of a side-on view and derived a similar equation for calculating the screen x coordinate as

x = w\*d/z

Where w = half the width of the road (from camera to road edge)

You can see that for both x and y, what we are really doing is scaling by a factor of

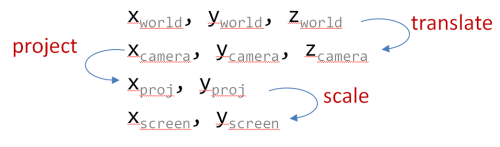
d/z

## Coordinate Systems

This sounds nice and simple in diagram form, but once you start coding its easy to get a little confused because we have been a bit loose in naming our variables and its not clear which represent 3d world coordinates and which represent 2d screen coordinates. We’ve also assumed that the camera is at the origin of our world when in reality it will be following our car.

More formally we should be:

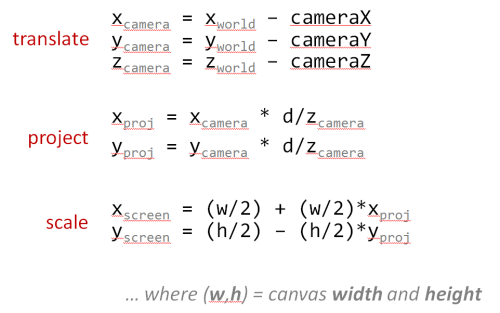
1. translating from world coordinates to camera coordinates
2. projecting camera coordinates onto a normalized projection plane
3. scaling the projected coordinates to physical screen (in our case canvas) coordinates



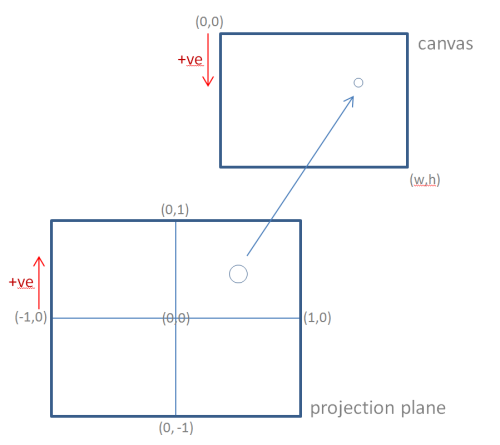
*NOTE: in a true 3d system a rotation step would come between steps 1 and 2, but since we’re going to be faking curves we dont need to worry about rotation*

## Projection

And so we can present our formal projection equations as follows:

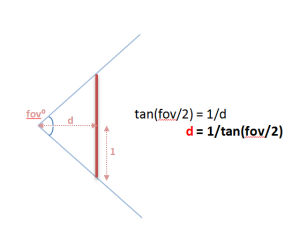


* The translate equations calculate the point relative to the camera
* The project equations are variations of our ‘law of similar triangles’ above
* The scale equations take into account the difference between:
  + *math* - where 0,0 is at the center and the y axis goes up and
  + *computers* - where 0,0 is at the top-left and the y axis goes down, as shown below:



*NOTE: In a full blown 3d system we would more formally define a Vector and a Matrixclass to perform more robust 3d mathematics, and if we were going to do that then we might as well just use WebGL (or equivalent)… but thats not really the point of this project. I really wanted to stick to old-school ‘just-enough’ pseudo-3d to build an outrun-style game.*

## Some More Trigonometry



One last piece of the puzzle is how to calculate d - the distance from the camera to the projection plane.

Instead of hard coding a value for d, its more useful to derive it from the desired vertical field of view. This way we can choose to ‘zoom’ the camera if needed.

Assuming we are projecting onto a normalized projection plane, with coordinates from -1 to +1, we can calculate d as follows:

d = 1/tan(fov/2)

Setting up fov as one (of many) variables we will be able to tweak in order to fine tune the rendering algorithm.

## Javascript Code Structure

I mentioned in the introduction that this code does not exactly follow javascript best practices - its a quick and dirty demo with simple global variables and functions. However, since I am going to build 4 separate versions (straights, curves, hills and sprites) I will keep some re-usable methods inside common.js within the following modules:

* Dom - a few minor DOM helpers.
* Util - generic utilities, mostly math helpers.
* Game - generic game helpers such as an image loader and the game loop.
* Render - canvas rendering helpers.

I will only be detailing methods from common.js if they are relevent to the actual game, rather than just simple DOM or math helpers. Hopefully you can tell from the name and context what the methods are supposed to do.

*As usual, the source code is the final documentation.*

## A Simple Game Loop

I won’t go into much detail here, I’ll simply re-use some of the code from my previous games to come up with a fixed time step game loop using [requestAnimationFrame](http://paulirish.com/2011/requestanimationframe-for-smart-animating/)

The idea being that each of my 4 examples can call Game.run(...) and provide their own versions of

* update - the game world with a fixed time step.
* render - the game world whenever the browser allows.

run: function(options) {

Game.loadImages(options.images, function(images) {

var update = options.update, // method to update game logic is provided by caller

render = options.render, // method to render the game is provided by caller

step = options.step, // fixed frame step (1/fps) is specified by caller

now = null,

last = Util.timestamp(),

dt = 0,

gdt = 0;

function frame() {

now = Util.timestamp();

dt = Math.min(1, (now - last) / 1000); // using requestAnimationFrame have to be able to handle large delta's caused when it 'hibernates' in a background or non-visible tab

gdt = gdt + dt;

while (gdt > step) {

gdt = gdt - step;

update(step);

}

render();

last = now;

requestAnimationFrame(frame);

}

frame(); // lets get this party started

});

}

Again, this is a rehash of ideas from my previous canvas games, so if you need clarification on how the game loop works go back and read those earlier articles (or post a comment below!).

## Images and Sprites

Before our game loop starts, we load 2 separate sprite sheets:

* background - 3 parallax layers for sky, hills and trees
* sprites - the car sprites (plus trees and billboards to add to the final version)



The spritesheet was generated with a small Rake task using the sprite-factory Ruby Gem. This task generates the unified sprite sheets as well as the x,y,w,h coordinates to be stored in the BACKGROUND and SPRITES constants.

*NOTE: The backgrounds are home-made using Inkscape, while most of the sprites are placeholder graphics borrowed from the old genesis version of outrun and used here as teaching examples. If there are any pixel artists out there who want to provide original art to turn this into a real game please get in touch!*

## Game Variables

In addition to our background and sprite images we will need a number of game variables, including:

var fps = 60; // how many 'update' frames per second

var step = 1/fps; // how long is each frame (in seconds)

var width = 1024; // logical canvas width

var height = 768; // logical canvas height

var segments = []; // array of road segments

var canvas = Dom.get('canvas'); // our canvas...

var ctx = canvas.getContext('2d'); // ...and its drawing context

var background = null; // our background image (loaded below)

var sprites = null; // our spritesheet (loaded below)

var resolution = null; // scaling factor to provide resolution independence (computed)

var roadWidth = 2000; // actually half the roads width, easier math if the road spans from -roadWidth to +roadWidth

var segmentLength = 200; // length of a single segment

var rumbleLength = 3; // number of segments per red/white rumble strip

var trackLength = null; // z length of entire track (computed)

var lanes = 3; // number of lanes

var fieldOfView = 100; // angle (degrees) for field of view

var cameraHeight = 1000; // z height of camera

var cameraDepth = null; // z distance camera is from screen (computed)

var drawDistance = 300; // number of segments to draw

var playerX = 0; // player x offset from center of road (-1 to 1 to stay independent of roadWidth)

var playerZ = null; // player relative z distance from camera (computed)

var fogDensity = 5; // exponential fog density

var position = 0; // current camera Z position (add playerZ to get player's absolute Z position)

var speed = 0; // current speed

var maxSpeed = segmentLength/step; // top speed (ensure we can't move more than 1 segment in a single frame to make collision detection easier)

var accel = maxSpeed/5; // acceleration rate - tuned until it 'felt' right

var breaking = -maxSpeed; // deceleration rate when braking

var decel = -maxSpeed/5; // 'natural' deceleration rate when neither accelerating, nor braking

var offRoadDecel = -maxSpeed/2; // off road deceleration is somewhere in between

var offRoadLimit = maxSpeed/4; // limit when off road deceleration no longer applies (e.g. you can always go at least this speed even when off road)

Some of these can be adjusted using the tweak UI controls to allow you to vary some of the critical values at run-time to see what effect they have on the rendered road. Others are derived from the tweakable UI values and recalculated during the reset() method.

## Driving a Ferrari

We provide a key mapping to Game.run that allows for simple keyboard input that sets or clears variables to indicate any action the player is currently taking:

Game.run({

...

keys: [

{ keys: [KEY.LEFT, KEY.A], mode: 'down', action: function() { keyLeft = true; } },

{ keys: [KEY.RIGHT, KEY.D], mode: 'down', action: function() { keyRight = true; } },

{ keys: [KEY.UP, KEY.W], mode: 'down', action: function() { keyFaster = true; } },

{ keys: [KEY.DOWN, KEY.S], mode: 'down', action: function() { keySlower = true; } },

{ keys: [KEY.LEFT, KEY.A], mode: 'up', action: function() { keyLeft = false; } },

{ keys: [KEY.RIGHT, KEY.D], mode: 'up', action: function() { keyRight = false; } },

{ keys: [KEY.UP, KEY.W], mode: 'up', action: function() { keyFaster = false; } },

{ keys: [KEY.DOWN, KEY.S], mode: 'up', action: function() { keySlower = false; } }

],

...

}

The variables that manage the player’s state are:

* speed - the current speed.
* position - the current Z position down the track. Note this is actually the position of the camera, not the ferrari.
* playerX - the current X position across the road. Normalized from -1 to +1 to be independent of the actual roadWidth.

These variables are set within the update method, which will:

* update position based on current speed.
* update playerX if left or right arrow keys are pressed.
* accelerate speed if up arrow is pressed.
* decelerate speed if down arrow is pressed.
* decelerate speed if neither up or down arrows are pressed.
* decelerate speed if playerX is off the sides of the road and into the grass.

For straight roads, the update method is pretty clean and simple:

function update(dt) {

position = Util.increase(position, dt \* speed, trackLength);

var dx = dt \* 2 \* (speed/maxSpeed); // at top speed, should be able to cross from left to right (-1 to 1) in 1 second

if (keyLeft)

playerX = playerX - dx;

else if (keyRight)

playerX = playerX + dx;

if (keyFaster)

speed = Util.accelerate(speed, accel, dt);

else if (keySlower)

speed = Util.accelerate(speed, breaking, dt);

else

speed = Util.accelerate(speed, decel, dt);

if (((playerX < -1) || (playerX > 1)) && (speed > offRoadLimit))

speed = Util.accelerate(speed, offRoadDecel, dt);

playerX = Util.limit(playerX, -2, 2); // dont ever let player go too far out of bounds

speed = Util.limit(speed, 0, maxSpeed); // or exceed maxSpeed

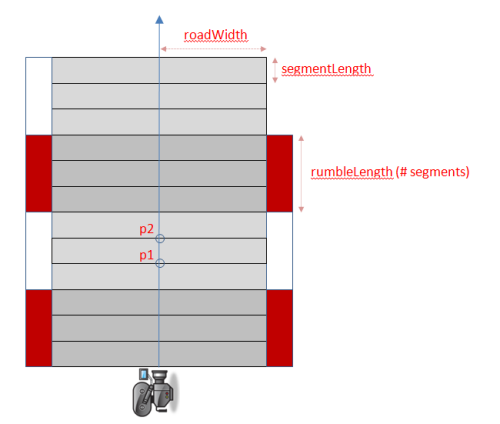
}

*Don’t worry, it will get much more complicated when we add sprites and collision detection in the final version :-)*

## Road Geometry

Before we can render our game world, we need to build up our array of road segments within the resetRoad() method.

Each of these road segments will eventually be projected from their world coordinates to become a 2d polygon in screen coordinates. We store 2 points for each segment, p1 is the center of the edge closest to the camera, while p2 is the center of the edge farthest from the camera.



Technically, each segments p2 is identical to the previous sections p1 but we will find it easier to maintain them as separate points and transform each segment independently.

The reason we maintain a separate rumbleLength is so that we can have fine detailed curves and hills but still have long rumble strips. If each alternating segment was a different color it would create a bad strobe effect. So we want lots of small segments, but group them together to form each rumble strip.

function resetRoad() {

segments = [];

for(var n = 0 ; n < 500 ; n++) { // arbitrary road length

segments.push({

index: n,

p1: { world: { z: n \*segmentLength }, camera: {}, screen: {} },

p2: { world: { z: (n+1)\*segmentLength }, camera: {}, screen: {} },

color: Math.floor(n/rumbleLength)%2 ? COLORS.DARK : COLORS.LIGHT

});

}

trackLength = segments.length \* segmentLength;

}

We initialize p1 and p2 with only z world coordinates because we only need straight roads. The ycoordinates will always be 0, while the x coordinates will always be based on a scaled+/- roadWidth. This will change later when we add curves and hills.

We also setup empty objects to store the camera and screen representations of these points to avoid creating lots of temporary objects during every render - trying to keep our garbage collection to a minimum we want to avoid allocating objects inside our game loop whenever possible.

When the car reaches the end of the road we will simply loop back to the beginning. To make this a little easier we provide a method to find the segment for any Z value even if it extends beyond the length of the road:

function findSegment(z) {

return segments[Math.floor(z/segmentLength) % segments.length];

}

## Rending the Background

Our render() method starts with drawing a background image. In future articles when we add curves and hills we will want the background to parallax scroll, so we start off in that direction here by rendering the background as 3 seperate layers:

function render() {

ctx.clearRect(0, 0, width, height);

Render.background(ctx, background, width, height, BACKGROUND.SKY);

Render.background(ctx, background, width, height, BACKGROUND.HILLS);

Render.background(ctx, background, width, height, BACKGROUND.TREES);

...

## Rending the Road

The render function then iterates over the segments, and projects each segment’s p1 and p2 from world coordinates to screen coordinates, clipping the segment if necessary, otherwise rendering it:

var baseSegment = findSegment(position);

var maxy = height;

var n, segment;

for(n = 0 ; n < drawDistance ; n++) {

segment = segments[(baseSegment.index + n) % segments.length];

Util.project(segment.p1, (playerX \* roadWidth), cameraHeight, position, cameraDepth, width, height, roadWidth);

Util.project(segment.p2, (playerX \* roadWidth), cameraHeight, position, cameraDepth, width, height, roadWidth);

if ((segment.p1.camera.z <= cameraDepth) || // behind us

(segment.p2.screen.y >= maxy)) // clip by (already rendered) segment

continue;

Render.segment(ctx, width, lanes,

segment.p1.screen.x,

segment.p1.screen.y,

segment.p1.screen.w,

segment.p2.screen.x,

segment.p2.screen.y,

segment.p2.screen.w,

segment.color);

maxy = segment.p2.screen.y;

}

We saw the math required to project a point earlier, the javascript version combines the translation, projection and scaling equations into a single method:

project: function(p, cameraX, cameraY, cameraZ, cameraDepth, width, height, roadWidth) {

p.camera.x = (p.world.x || 0) - cameraX;

p.camera.y = (p.world.y || 0) - cameraY;

p.camera.z = (p.world.z || 0) - cameraZ;

p.screen.scale = cameraDepth/p.camera.z;

p.screen.x = Math.round((width/2) + (p.screen.scale \* p.camera.x \* width/2));

p.screen.y = Math.round((height/2) - (p.screen.scale \* p.camera.y \* height/2));

p.screen.w = Math.round( (p.screen.scale \* roadWidth \* width/2));

}

In addition to calculating screen x and y for each of our p1 and p2 points we also use the same projection math to calculate the projected width (w) of the segment.

Given the screen x and y coordinates for both p1 and p2, along with the projected road width w, it becomes fairly straight forward for the Render.segment helper to calculate all the polygons it needs to render the grass, road, rumble strips and lane separators using a generic Render.polygon helper *(see common.js)*

## Rendering the Car

Finally, the last thing required by the render method is to render the ferrari:

Render.player(ctx, width, height, resolution, roadWidth, sprites, speed/maxSpeed,

cameraDepth/playerZ,

width/2,

height);

*The reason this method is named player instead of car is because our final version of the game has other cars on the road, and we want to specifically differentiate the player’s ferrari from the other cars.*

The Render.player helper ultimately uses the canvas drawImage method to render a sprite after scaling it based on the same projection scaling we saw earlier:

d/z

Where z in this case is the relative distance of the car from the camera stored in the playerZvariable.

It also ‘bounces’ the car a little at higher speeds by adding a little random-ness to the scaling equation based on speed/maxSpeed.

And boom there you have it:



## Conclusion

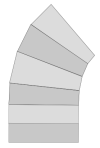
Thats actually a fairly large chunk of work already just to get us setup with straight roads. We added…

* a common Dom helper module
* a common Util math helper module
* a common Render canvas helper module…
* … including Render.segment, Render.polygon and Render.sprite
* a fixed step game loop
* an image loader
* a keyboard handler
* a parallax layered background
* a spritesheet full of cars, trees and billboards
* some rudimentary road geometry
* an update() method to drive a car
* a render() method to render background, road and player car
* an HTML5 <audio> tag with some racing music

… but it gives us a good foundation to build on.

# Curves

We only needed a z world coordinate for each point because, for straight roads, both x and ywere zero.



If we were building a full-blown 3d system we might implement curves by calculating x and z coordinates in a kind of fan-strip as shown on the left. However that kind of geometry can be a little complex to calculate and would require us to add a 3d rotation step to our projection equations…

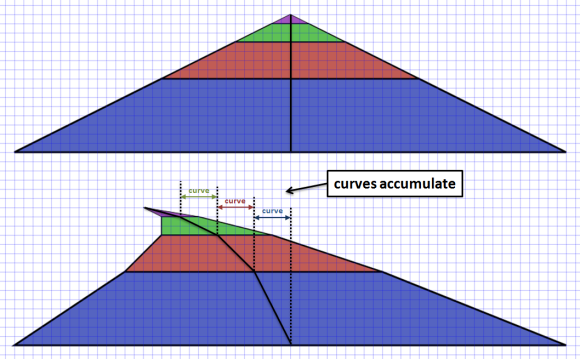
… if we wanted to go down that path we would be better off using WebGL or its equivalent, but that’s not really what this project is about. We just want to use some old-school ‘good enough’ pseudo 3d tricks to fake our curves.

So you might be surprised to learn that we wont be calculating x coordinates for our road segments at all…

Instead we’ll follow [Lou’s advice](http://www.extentofthejam.com/pseudo/):

*“To curve a road, you just need to change the position of the center-line in a curve shape… starting at the bottom of the screen, the amount that the center of the road shifts left or right steadily increases”*

In our case, the center-line is the cameraX value we pass to the projection calculations. This means that as we render() each segment of the road, we can fake curves by offsetting the cameraXvalue by a steadily increasing amount.



In order to know how much to offset we need to store a curve value in each segment. This value represents how much the segment should be offset from the camera’s center line, and will be:

* negative for left hand curves
* positive for right hand curves
* smaller for easy curves
* larger for harder curves

The actual values are somewhat arbitrary, and through trial and error we can find good values to make our curves ‘feel’ right:

var ROAD = {

LENGTH: { NONE: 0, SHORT: 25, MEDIUM: 50, LONG: 100 }, // num segments

CURVE: { NONE: 0, EASY: 2, MEDIUM: 4, HARD: 6 }

};

In addition to defining good curve values. We want to avoid any jarring transitions when a straight turns into a curve (or vice versa) by [easing](http://jqueryui.com/demos/effect/easing.html) into and out of the curves. We do this by slowly incrementing (or decrementing) the curve value for each segment until it reaches our desired value using traditional easing functions such as:

easeIn: function(a,b,percent) { return a + (b-a)\*Math.pow(percent,2); },

easeOut: function(a,b,percent) { return a + (b-a)\*(1-Math.pow(1-percent,2)); },

easeInOut: function(a,b,percent) { return a + (b-a)\*((-Math.cos(percent\*Math.PI)/2) + 0.5); },

So now, given a function to add a single segment to our geometry…

function addSegment(curve) {

var n = segments.length;

segments.push({

index: n,

p1: { world: { z: n \*segmentLength }, camera: {}, screen: {} },

p2: { world: { z: (n+1)\*segmentLength }, camera: {}, screen: {} },

curve: curve,

color: Math.floor(n/rumbleLength)%2 ? COLORS.DARK : COLORS.LIGHT

});

}

… we can create a method to ease in, hold, and then ease out of a curved road:

function addRoad(enter, hold, leave, curve) {

var n;

for(n = 0 ; n < enter ; n++)

addSegment(Util.easeIn(0, curve, n/enter));

for(n = 0 ; n < hold ; n++)

addSegment(curve);

for(n = 0 ; n < leave ; n++)

addSegment(Util.easeInOut(curve, 0, n/leave));

}

… and we can layer additional geometry on top, such as S-Curves:

function addSCurves() {

addRoad(ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, -ROAD.CURVE.EASY);

addRoad(ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.CURVE.MEDIUM);

addRoad(ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.CURVE.EASY);

addRoad(ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, -ROAD.CURVE.EASY);

addRoad(ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, ROAD.LENGTH.MEDIUM, -ROAD.CURVE.MEDIUM);

}

## Changes to the update() method

The only changes we need to make to our existing update() method is to apply some kind of centrifugal force to the car when its going around the curve.

We define an arbitrary multiplier that can be tuned until it ‘feels good’

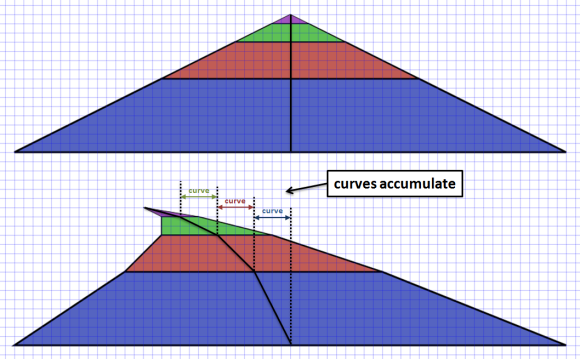
var centrifugal = 0.3; // centrifugal force multiplier when going around curves

And we simply update the playerX position based on their current speed, the current curve amount and the centrifugal force multiplier:

playerX = playerX - (dx \* speedPercent \* playerSegment.curve \* centrifugal);

## Rendering curves

Earlier we said we could render fake curves by offsetting the cameraX value used in the projection calculations as we render() each segment of the road.



To do that we maintain an accumulating dx variable that increases by the amount of curve for each segment, along with an x variable that will be used as the offset to the cameraX value used in the projection calculations.

To implement curves we need to:

* offset each segments p1 projection by x
* offset each segments p2 projection by x + dx
* increase x by dx for the next segment

Finally, in order to avoid jarring transitions when crossing segment boundaries we must ensure dxis initialized with an interpolated value for the current base segments curve.

Modifying our render() method like so:

var baseSegment = findSegment(position);

var basePercent = Util.percentRemaining(position, segmentLength);

var dx = - (baseSegment.curve \* basePercent);

var x = 0;

for(n = 0 ; n < drawDistance ; n++) {

...

Util.project(segment.p1, (playerX \* roadWidth) - x, cameraHeight, position - (segment.looped ? trackLength : 0), cameraDepth, width, height, roadWidth);

Util.project(segment.p2, (playerX \* roadWidth) - x - dx, cameraHeight, position - (segment.looped ? trackLength : 0), cameraDepth, width, height, roadWidth);

x = x + dx;

dx = dx + segment.curve;

...

}

*Hmmm. If I was brutally honest, I’d have to admit that this made a lot more sense when I was writing the code than it does now trying to explain it for others. Looking back now it looks suspiciously like I have a double accumulation going on and I can’t really justify the need for both x and dx ? That’s a terrible admission as a programmer!!… You know what, forget I said anything, there’s nothing to see here, pretend you didn’t read this note and lets move on…*

*UPDATE. Thanks to PeteB in comments below for reminding me that a curve is a 2nd order equation, and that I do need to maintain a separate dx as the rate of change of x. I started second guessing myself when writing this article, and I was also in a dazed and confused state of mind due to England getting knocked out of Euro2012 - on penalties - AGAIN! So its ok, there was nothing to worry about, this code is correct (at least as correct as a ‘fake’ curve can be!)*

## Parallax scrolling background

Finally, we need to scroll the parallax background layers by maintaining an offset for each layer…

var skySpeed = 0.001; // background sky layer scroll speed when going around curve (or up hill)

var hillSpeed = 0.002; // background hill layer scroll speed when going around curve (or up hill)

var treeSpeed = 0.003; // background tree layer scroll speed when going around curve (or up hill)

var skyOffset = 0; // current sky scroll offset

var hillOffset = 0; // current hill scroll offset

var treeOffset = 0; // current tree scroll offset

… and increasing it during update() based on the curve of the players current segment and their speed…

skyOffset = Util.increase(skyOffset, skySpeed \* playerSegment.curve \* speedPercent, 1);

hillOffset = Util.increase(hillOffset, hillSpeed \* playerSegment.curve \* speedPercent, 1);

treeOffset = Util.increase(treeOffset, treeSpeed \* playerSegment.curve \* speedPercent, 1);

… and then use that offset when we render() the background layers

Render.background(ctx, background, width, height, BACKGROUND.SKY, skyOffset);

Render.background(ctx, background, width, height, BACKGROUND.HILLS, hillOffset);

Render.background(ctx, background, width, height, BACKGROUND.TREES, treeOffset);

## Conclusion

So there you have it, fake psuedo-3d curves:



Most of the code we added for curves revolves around constructing the road geometry with the appropriate curve value. Once we have that, providing a centrifugal force during update() is easy.

Rendering the curves is only a few lines of code, but they can be conceptually hard to understand (and describe) exactly what’s happening. There are many approaches to faking curves and its easy to go down some dead ends, and even easier to get side tracked trying to do the ‘correct’ thing and before you know it you are implementing a full blown 3d system with matrices, rotation and true 3-d geometry… which I’ve already said is not the point here.

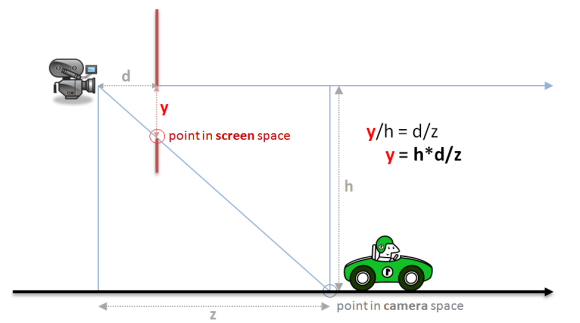
In writing this article I’m pretty sure that I actually have some problems with my curve implementation. In trying to visualize the algorithm for this article I can’t help but wonder why I need 2 accumulating values dx and x instead of just one… and if I’m not able to fully explain it then something, somewhere is wrong…

… but my time on this *‘weekend’* (ha!) project is pretty much up and, to be honest, the curves look pretty good to me - and really that’s what matters at the end of the day.

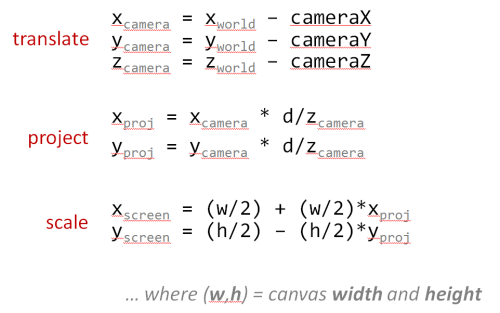
# Hills

This time around we’re going to tackle hills, and luckily it’s a lot easier than when we included curved roads.

In the earlier article we used a law of similar triangles to introduce 3d perspective projection:

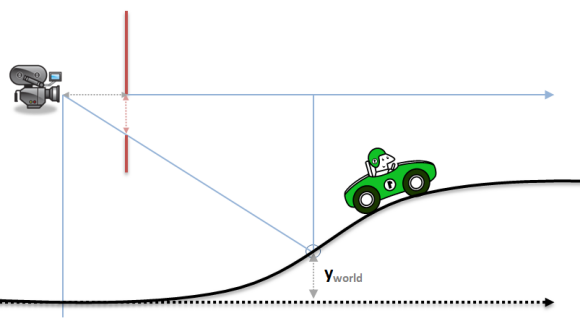


… which led us to our equations for projecting a 3d world coordinate into a 2d screen coordinate.



… but since we were only dealing with straight roads at the time our world coordinates only needed a z component because both x and y were zero.

This set us up nicely because now, to add hills into the mix, all we need to do is give our road segments an appropriate non-zero y coordinate and our existing render() function will just magically work.



Yup. That’s really all we have to do to get hills. Add a y component to each road segments world coordinates.

## Changes to the road geometry

We modify our existing addSegment method to allow p2.world.y to be provided by the caller, while p1.world.y is set to match p2.world.y of the immediately preceding segment:

function addSegment(curve, y) {

var n = segments.length;

segments.push({

index: n,

p1: { world: { y: lastY(), z: n \*segmentLength }, camera: {}, screen: {} },

p2: { world: { y: y, z: (n+1)\*segmentLength }, camera: {}, screen: {} },

curve: curve,

color: Math.floor(n/rumbleLength)%2 ? COLORS.DARK : COLORS.LIGHT

});

}

function lastY() {

return (segments.length == 0) ? 0 : segments[segments.length-1].p2.world.y;

}

Add some constants to represent LOW, MEDIUM or HIGH hills:

var ROAD = {

LENGTH: { NONE: 0, SHORT: 25, MEDIUM: 50, LONG: 100 },

HILL: { NONE: 0, LOW: 20, MEDIUM: 40, HIGH: 60 },

CURVE: { NONE: 0, EASY: 2, MEDIUM: 4, HARD: 6 }

};

Modify our existing addRoad() method to accept a y argument that will be used, along with easing functions, to gradually ease into and out of the hill:

function addRoad(enter, hold, leave, curve, y) {

var startY = lastY();

var endY = startY + (Util.toInt(y, 0) \* segmentLength);

var n, total = enter + hold + leave;

for(n = 0 ; n < enter ; n++)

addSegment(Util.easeIn(0, curve, n/enter), Util.easeInOut(startY, endY, n/total));

for(n = 0 ; n < hold ; n++)

addSegment(curve, Util.easeInOut(startY, endY, (enter+n)/total));

for(n = 0 ; n < leave ; n++)

addSegment(Util.easeInOut(curve, 0, n/leave), Util.easeInOut(startY, endY, (enter+hold+n)/total));

}

Finally, just like we did in the previous article with addSCurves(), we can layer on whatever geometry building methods we like, such as:

function addLowRollingHills(num, height) {

num = num || ROAD.LENGTH.SHORT;

height = height || ROAD.HILL.LOW;

addRoad(num, num, num, 0, height/2);

addRoad(num, num, num, 0, -height);

addRoad(num, num, num, 0, height);

addRoad(num, num, num, 0, 0);

addRoad(num, num, num, 0, height/2);

addRoad(num, num, num, 0, 0);

}

## Changes to the update method

In an arcade game like this, where we are not attempting to simulate reality, the hills don’t affect the player or the game world in any real way so there are no changes required for the update()method.

## Rendering hills

… and there are also no changes required for the render() method either since our projection equations were already set up to project our road segments correctly with non-zero y coordinates from the start.

## Parallax scrolling background

Apart from adding y coordinates to all the road segments, the only other change we might make is to ensure the background layers are offset vertically with the hills (as well as horizontally with the curves). We do this with the addition of a final argument to the Render.background helper.

The simplest mechanism is to simply offset it relative to the playerY position (which needs to be interpolated from the world y positions of the players current segment).

This isn’t the most realistic behavior because we probably should take into account the slope of the players current road segment, but the effect is simple and works well enough for a simple demo.

## Conclusion

So there you have it, we can now add real hills to our (fake) curves:



# Conclusion

now we’re into the final lap! *(groan)*. In this article we will add:

* Billboards and trees
* Other cars
* Collision detection
* Rudimentary car AI
* A HUD with lap timer and fastest lap

… which will give us just enough interactivity to justify finally calling this a ‘game’.

## A note on code structure

*I mentioned early on in this series that this is code for a tech demo with global variables and few classes/structures, it is certainly not an example of javascript best practices.*

*In this final section, as we add much more code for maintaining sprites and updating other cars, we cross that (subjective) boundary where previously simple code starts to become more complex and could benefit from a little more structure…*

*… but it wont get that structure because its just a tech demo, so please forgive the slightly messy code examples in this article, and trust that in a real project we would want to aggressively clean this up.*

## Sprites



In part 1, before our game loop started, we loaded a sprite sheet containing all of the cars, trees, and billboards.

You could create a sprite sheet manually in any image editor, but maintaining the images and calculating the coordinates is best done with an automated tool. In this case, my spritesheet was generated with a small Rake task using the sprite-factory Ruby Gem.

This task generates the unified sprite sheets from individual image files as well as calculating the x,y,w,h coordinates to be stored in a SPRITESconstant:

var SPRITES = {

PALM\_TREE: { x: 5, y: 5, w: 215, h: 540 },

BILLBOARD08: { x: 230, y: 5, w: 385, h: 265 },

// ... etc

CAR04: { x: 1383, y: 894, w: 80, h: 57 },

CAR01: { x: 1205, y: 1018, w: 80, h: 56 },

};

## Adding billboards and trees

We add an array to each road segment to contain our roadside sprites.

Each sprite consists of a source from the SPRITES collection along with a horizontal offsetwhich is normalized so that -1 indicates the left edge of the road while +1 indicates the right edge of the road, allowing us to stay independent of the actual roadWidth.

Some sprites are placed very deliberately, while others are randomized.

function addSegment() {

segments.push({

...

sprites: [],

...

});

}

function addSprite(n, sprite, offset) {

segments[n].sprites.push({ source: sprite, offset: offset });

}

function resetSprites() {

addSprite(20, SPRITES.BILLBOARD07, -1);

addSprite(40, SPRITES.BILLBOARD06, -1);

addSprite(60, SPRITES.BILLBOARD08, -1);

addSprite(80, SPRITES.BILLBOARD09, -1);

addSprite(100, SPRITES.BILLBOARD01, -1);

addSprite(120, SPRITES.BILLBOARD02, -1);

addSprite(140, SPRITES.BILLBOARD03, -1);

addSprite(160, SPRITES.BILLBOARD04, -1);

addSprite(180, SPRITES.BILLBOARD05, -1);

addSprite(240, SPRITES.BILLBOARD07, -1.2);

addSprite(240, SPRITES.BILLBOARD06, 1.2);

for(n = 250 ; n < 1000 ; n += 5) {

addSprite(n, SPRITES.COLUMN, 1.1);

addSprite(n + Util.randomInt(0,5), SPRITES.TREE1, -1 - (Math.random() \* 2));

addSprite(n + Util.randomInt(0,5), SPRITES.TREE2, -1 - (Math.random() \* 2));

}

...

}

*NOTE: If we were building a real game we would want to build a road editor of some kind for visually creating a map with hills and curves and include a mechanism to place sprites along the road…. but for our purposes we can simply addSprite()programatically.*

## Adding cars

In addition to our roadside sprites, we add a collection of the cars that occupy each segment. Along with a separate collection of all the cars on the track.

var cars = []; // array of cars on the road

var totalCars = 200; // total number of cars on the road

function addSegment() {

segments.push({

...

cars: [], // array of cars within this segment

...

});

}

Maintaining 2 data structures for cars allows us to easily iterate over all the cars during the update() method, moving them from one segment to another as necessary, while at the same time allowing us to only render() the cars on the visible segments.

Each car is given a random horizontal offset, z position, sprite source and speed:

function resetCars() {

cars = [];

var n, car, segment, offset, z, sprite, speed;

for (var n = 0 ; n < totalCars ; n++) {

offset = Math.random() \* Util.randomChoice([-0.8, 0.8]);

z = Math.floor(Math.random() \* segments.length) \* segmentLength;

sprite = Util.randomChoice(SPRITES.CARS);

speed = maxSpeed/4 + Math.random() \* maxSpeed/(sprite == SPRITES.SEMI ? 4 : 2);

car = { offset: offset, z: z, sprite: sprite, speed: speed };

segment = findSegment(car.z);

segment.cars.push(car);

cars.push(car);

}

}

## Rendering hills (revisited)

In previous articles I talked about rendering road segments, including curves and hills, but there were a few lines of code I glossed over regarding a maxy variable that started off at the bottom of the screen, but decreased as we rendered each segment to indicate how much of the screen had already been rendered:

for(n = 0 ; n < drawDistance ; n++) {

...

if ((segment.p1.camera.z <= cameraDepth) || // behind us

(segment.p2.screen.y >= maxy)) // clip by (already rendered) segment

continue;

...

maxy = segment.p2.screen.y;

}

This allows us to clip segments that would be obscured by already rendered hills.

A traditional [painters algorithm](http://en.wikipedia.org/wiki/Painter's_algorithm) would normally render from back to front, having the nearer segments overwrite the further segments. However we can’t afford to waste time rendering polygons that will ultimately get overwritten, so it becomes easier to render front to back and clip far segments that have been obscured by already rendered near segments if their projected coordinates are lower than maxy.

## Rendering billboards, trees and cars

However, iterating over the road segments front to back will not work when rendering sprites because they frequently overlap and therefore must be rendered back to front using the painters algorithm.

This complicates our render() method and forces us to loop over the road segments in two phases:

1. front to back to render the road
2. back to front to render the sprites



In addition to having to deal with sprites that overlap, we also need to deal with sprites that are ‘just over’ a hilltop horizon. If the sprite is tall enough we should be able to see the top of it even if the road segment it sits in is on the backside of the hill and therefore not rendered.

We can tackle this latter problem by saving the maxyvalue for each segment as a clip line during phase 1. Then we can clip the sprites on that segment to the clipline during phase 2.

The remainder of the rendering logic figures out how much to scale and position the sprite, based on the road segments scale factor and screen coordinates (calculated in phase 1), leaving us with something like this for the second phase of our render() method:

// back to front painters algorithm

for(n = (drawDistance-1) ; n > 0 ; n--) {

segment = segments[(baseSegment.index + n) % segments.length];

// render roadside sprites

for(i = 0 ; i < segment.sprites.length ; i++) {

sprite = segment.sprites[i];

spriteScale = segment.p1.screen.scale;

spriteX = segment.p1.screen.x + (spriteScale \* sprite.offset \* roadWidth \* width/2);

spriteY = segment.p1.screen.y;

Render.sprite(ctx, width, height, resolution, roadWidth, sprites, sprite.source, spriteScale, spriteX, spriteY, (sprite.offset < 0 ? -1 : 0), -1, segment.clip);

}

// render other cars

for(i = 0 ; i < segment.cars.length ; i++) {

car = segment.cars[i];

sprite = car.sprite;

spriteScale = Util.interpolate(segment.p1.screen.scale, segment.p2.screen.scale, car.percent);

spriteX = Util.interpolate(segment.p1.screen.x, segment.p2.screen.x, car.percent) + (spriteScale \* car.offset \* roadWidth \* width/2);

spriteY = Util.interpolate(segment.p1.screen.y, segment.p2.screen.y, car.percent);

Render.sprite(ctx, width, height, resolution, roadWidth, sprites, car.sprite, spriteScale, spriteX, spriteY, -0.5, -1, segment.clip);

}

}

## Colliding with billboards and trees

So now that we can add, and render, roadside sprites, we need to modify our update() method to detect if the player has collided with any of the sprites in the players current segment:

We use a helper method Util.overlap() to provide a generic rectangle overlap detection and if an overlap is detected we stop the car:

if ((playerX < -1) || (playerX > 1)) {

for(n = 0 ; n < playerSegment.sprites.length ; n++) {

sprite = playerSegment.sprites[n];

spriteW = sprite.source.w \* SPRITES.SCALE;

if (Util.overlap(playerX, playerW, sprite.offset + spriteW/2 \* (sprite.offset > 0 ? 1 : -1), spriteW)) {

// stop the car

break;

}

}

}

*NOTE: if you examine the real code you see that we dont actually stop the car because then they can no longer drive sideways around the obstacle, as a hack shortcut we keep their position fixed and allow the car to ‘slide’ sideways around the sprite.*

## Colliding with Cars

In addition to colliding with road side sprites, we need to detect collision with other cars, and if an overlap is detected we slow the player down and ‘bounce’ back behind the car that we collided with:

for(n = 0 ; n < playerSegment.cars.length ; n++) {

car = playerSegment.cars[n];

carW = car.sprite.w \* SPRITES.SCALE;

if (speed > car.speed) {

if (Util.overlap(playerX, playerW, car.offset, carW, 0.8)) {

// slow the car

break;

}

}

}

## Updating Cars

To make the other cars drive along the road, we give them the simplest possible AI:

* drive at a constant speed
* automatically steer around the player when overtaking
* automatically steer around other cars when overtaking

*NOTE: we don’t actually have to worry about steering other cars around curves in the road because the curves are fake. If we just keep the cars moving forward through our road segments they will automatically make it through the curves.*

This all happens during the game update() loop with a call to updateCars() where we move each car forward at constant speed and switch from one segment to the next if they have driven far enough during this frame.

function updateCars(dt, playerSegment, playerW) {

var n, car, oldSegment, newSegment;

for(n = 0 ; n < cars.length ; n++) {

car = cars[n];

oldSegment = findSegment(car.z);

car.offset = car.offset + updateCarOffset(car, oldSegment, playerSegment, playerW);

car.z = Util.increase(car.z, dt \* car.speed, trackLength);

car.percent = Util.percentRemaining(car.z, segmentLength); // useful for interpolation during rendering phase

newSegment = findSegment(car.z);

if (oldSegment != newSegment) {

index = oldSegment.cars.indexOf(car);

oldSegment.cars.splice(index, 1);

newSegment.cars.push(car);

}

}

}

The updateCarOffset() method provides the *‘AI’* that allows a car to steer around the player or another car. It is one of the more complex methods in the entire code base, and for a real game would need to become a lot more complex to make the cars appear more realistic than they do in this simple demo.

For our purposes we will stick to a naive, brute-force AI and have each car

* lookahead 20 segments
* if it detects a slower car ahead that overlaps, then steer around it
* steer to the right of obstacles that are on the left side of the road
* steer to the left of obstacles that are on the right side of the road
* steer enough to avoid the obstacle ahead within the distance remaining

We can also cheat with cars that are not visible to the player and simply skip any steering, letting those cars overlap and drive through each other. They only need to ‘be smart’ when in view of the player.

function updateCarOffset(car, carSegment, playerSegment, playerW) {

var i, j, dir, segment, otherCar, otherCarW, lookahead = 20, carW = car.sprite.w \* SPRITES.SCALE;

// optimization, dont bother steering around other cars when 'out of sight' of the player

if ((carSegment.index - playerSegment.index) > drawDistance)

return 0;

for(i = 1 ; i < lookahead ; i++) {

segment = segments[(carSegment.index+i)%segments.length];

if ((segment === playerSegment) && (car.speed > speed) && (Util.overlap(playerX, playerW, car.offset, carW, 1.2))) {

if (playerX > 0.5)

dir = -1;

else if (playerX < -0.5)

dir = 1;

else

dir = (car.offset > playerX) ? 1 : -1;

return dir \* 1/i \* (car.speed-speed)/maxSpeed; // the closer the cars (smaller i) and the greater the speed ratio, the larger the offset

}

for(j = 0 ; j < segment.cars.length ; j++) {

otherCar = segment.cars[j];

otherCarW = otherCar.sprite.w \* SPRITES.SCALE;

if ((car.speed > otherCar.speed) && Util.overlap(car.offset, carW, otherCar.offset, otherCarW, 1.2)) {

if (otherCar.offset > 0.5)

dir = -1;

else if (otherCar.offset < -0.5)

dir = 1;

else

dir = (car.offset > otherCar.offset) ? 1 : -1;

return dir \* 1/i \* (car.speed-otherCar.speed)/maxSpeed;

}

}

}

}

This algorithm works well enough in most cases, but if there is heavy traffic ahead we might find our cars rubberbanding left and then right as they try to squeeze through a gap between 2 other vehicles. There are many ways we can make this AI more robust, one of which might be to allow the cars to slow down when there is not enough room to steer around obstacles.

## Heads up display

And finally, we add a rudimentary HTML heads up display (HUD):

<div id="hud">

<span id="speed" class="hud"><span id="speed\_value" class="value">0</span> mph</span>

<span id="current\_lap\_time" class="hud">Time: <span id="current\_lap\_time\_value" class="value">0.0</span></span>

<span id="last\_lap\_time" class="hud">Last Lap: <span id="last\_lap\_time\_value" class="value">0.0</span></span>

<span id="fast\_lap\_time" class="hud">Fastest Lap: <span id="fast\_lap\_time\_value" class="value">0.0</span></span>

</div>

… style it with CSS:

#hud { position: absolute; z-index: 1; width: 640px; padding: 5px 0; font-family: Verdana, Geneva, sans-serif; font-size: 0.8em; background-color: rgba(255,0,0,0.4); color: black; border-bottom: 2px solid black; box-sizing: border-box; -moz-box-sizing: border-box; -webkit-box-sizing: border-box; }

#hud .hud { background-color: rgba(255,255,255,0.6); padding: 5px; border: 1px solid black; margin: 0 5px; transition-property: background-color; transition-duration: 2s; -webkit-transition-property: background-color; -webkit-transition-duration: 2s; }

#hud #speed { float: right; }

#hud #current\_lap\_time { float: left; }

#hud #last\_lap\_time { float: left; display: none; }

#hud #fast\_lap\_time { display: block; width: 12em; margin: 0 auto; text-align: center; transition-property: background-color; transition-duration: 2s; -webkit-transition-property: background-color; -webkit-transition-duration: 2s; }

#hud .value { color: black; font-weight: bold; }

#hud .fastest { background-color: rgba(255,215,0,0.5); }

… and update() it during our game loop:

if (position > playerZ) {

if (currentLapTime && (startPosition < playerZ)) {

lastLapTime = currentLapTime;

currentLapTime = 0;

if (lastLapTime <= Util.toFloat(Dom.storage.fast\_lap\_time)) {

Dom.storage.fast\_lap\_time = lastLapTime;

updateHud('fast\_lap\_time', formatTime(lastLapTime));

Dom.addClassName('fast\_lap\_time', 'fastest');

Dom.addClassName('last\_lap\_time', 'fastest');

}

else {

Dom.removeClassName('fast\_lap\_time', 'fastest');

Dom.removeClassName('last\_lap\_time', 'fastest');

}

updateHud('last\_lap\_time', formatTime(lastLapTime));

Dom.show('last\_lap\_time');

}

else {

currentLapTime += dt;

}

}

updateHud('speed', 5 \* Math.round(speed/500));

updateHud('current\_lap\_time', formatTime(currentLapTime));

Our updateHud() helper method allows us to only update DOM elements only if the value has changed because updating DOM elements can be slow and we shouldn’t do it 60fps unless the actual values have changed.

function updateHud(key, value) { // accessing DOM can be slow, so only do it if value has changed

if (hud[key].value !== value) {

hud[key].value = value;

Dom.set(hud[key].dom, value);

}

}

## Conclusion



Phew! That was a long last lap, but there you have it, our final version has entered that stage where it can legitimately be called a game. It’s still very far from a finished game, but it’s a game nonetheless.