Case Study - Building Racer

- The Track
  - Calculating Track Dimensions
  - Drawing the Track
- CSS Animations
- CSS Sprites
- Rendering the Cars
- Keeping the Devices Synced
  - Calculating Latency
  - Acceleration/Deceleration
  - Keeping the Cars Synced
- Conclusion

Racer is a web-based mobile Chrome Experiment developed by Active Theory. Up to 5 friends can connect their phones or tablets to race across every screen. Armed with the concept, design and prototype from Google Creative Lab and sound from Plan8 we iterated on builds for 8 weeks leading up to the launch at I/O 2013. Now that the game has been live for a few weeks we've had the chance to field some questions from the developer community on how it works. Below is a breakdown of the key features and answers to the questions we're most often asked.

The Track

A fairly obvious challenge we faced was how to make a web based mobile game that works well across a wide variety of devices. Players needed to be able to construct a race with different phones and tablets. One player could have a Nexus 4 and want to race against his friend who had an iPad. We needed to come up with a way to determine a common track size for each race. The solution had to involve using different size tracks depending on the specs for each device included in the race.

Calculating Track Dimensions

As each player joins, information about their device is sent to the server and shared with other players. When the track is being built, this data is used to calculate the height and width of the track. We calculate the height by finding the height of the smallest screen, and the width is the total width of all screens. So in the example below the track would have a width of 1152 pixels and a height of 519 pixels.

The red area shows the total width and height of the track for this example.

The red area shows the total width and height of the track for this example.

```
this.getDimensions = function () {
  var response = {};
  response.width = 0;
  response.height = _gamePlayers[0].scrn.h; // First screen height
  response.screens = [];

  for (var i = 0; i < _gamePlayers.length; i++) {
    var player = _gamePlayers[i];
    response.width += player.scrn.w;

    if (player.scrn.h < response.height) {
      // Find the smallest screen height
      response.height = player.scrn.h;
    }

    response.screens.push(player.scrn);
  }

  return response;
}
```

Drawing the Track

Paper.js is an open source vector graphics scripting framework that runs on top of HTML5 Canvas. We found Paper.js was the perfect tool to create vector shapes for the tracks, so we used its capabilities to render the SVG tracks that were built in Adobe Illustrator on a `<canvas>` element. To create the track, the `TrackModel` class appends the SVG code to the DOM and gathers information about the original dimensions and positioning to be passed to the `TrackPathView` which will draw the track to a canvas.

```
paper.install(window);
_paper = new paper.PaperScope();
_paper.setup('track_canvas');

var svg = document.getElementById('track');
var layer = new _paper.Layer();

_path = layer.importSvg(svg).firstChild.firstChild;
_path.strokeColor = '#14a8df';
_path.strokeWidth = 2;
```

Once the track is drawn, each device finds its x offset based on it’s position in the device line-up order, and positions the track accordingly.

```
var x = 0;

for (var i = 0; i < screens.length; i++) {
  if (i < PLAYER_INDEX) {
    x += screens[i].w;
  }
}
```

> *The x offset can then be used to show the appropriate portion of the track.*
>
> The x offset can then be used to show the appropriate portion of the track

CSS Animations

Paper.js uses a lot of CPU processing to draw the track lanes and this process will take more or less time on different devices. To handle this, we needed a loader to loop until all devices finish processing the track. The problem was that any JavaScript-based animation would skip frames due to the CPU requirements of Paper.js. Enter CSS animations, which run on a separate UI thread, allowing us to smoothly animate the sheen across the "BUILDING TRACK" text.

```
.glow {
  width: 290px;
  height: 290px;
  background: url('img/track-glow.png') 0 0 no-repeat;
  background-size: 100%;
```

```
  top: 0;
  left: -290px;
  z-index: 1;
  -webkit-animation: wipe 1.3s linear 0s infinite;
}

@-webkit-keyframes wipe {
  0% {
    -webkit-transform: translate(-300px, 0);
  }

  25% {
    -webkit-transform: translate(-300px, 0);
  }

  75% {
    -webkit-transform: translate(920px, 0);
  }

  100% {
    -webkit-transform: translate(920px, 0);
  }
}
}
```

CSS Sprites

CSS also came in handy for in-game effects. Mobile devices, with their limited power, are kept busy animating the cars running across the tracks. So for additional excitement we used sprites as a way of implementing pre-rendered animations into the game. In a CSS sprite, transitions apply a step-based animation which changes the `background-position` property, creating the car explosion.

```
#sprite {
  height: 100px;
  width: 100px;
  background: url('sprite.jpg') 0 0 no-repeat;
  -webkit-animation: play-sprite 0.33s linear 0s steps(9) infinite;
}
```

```
@-webkit-keyframes play-sprite {
  0% {
    background-position: 0 0;
  }

  100% {
    background-position: -900px 0;
  }
}
```

The problem with this technique is you can only use sprite sheets laid out on a single row. In order to loop through multiple rows, the animation must be chained through multiple keyframe declarations.

```
#sprite {
  height: 100px;
  width: 100px;
  background: url('sprite.jpg') 0 0 no-repeat;
  -webkit-animation-name: row1, row2, row3;
  -webkit-animation-duration: 0.2s;
  -webkit-animation-delay: 0s, 0.2s, 0.4s;
  -webkit-animation-timing-function: steps(5), steps(5), steps(5);
  -webkit-animation-fill-mode: forwards;
}

@-webkit-keyframes row1 {
  0% {
    background-position: 0 0;
  }

  100% {
    background-position: -500px 0;
  }
}

@-webkit-keyframes row2 {
  0% {
    background-position: 0 -100px;
  }
```

```
  100% {
    background-position: -500px -100px;
  }
}

@-webkit-keyframes row3 {
  0% {
    background-position: 0 -200px;
  }

  100% {
    background-position: -500px -200px;
  }
}
```

Rendering the Cars

As with any car racing game we knew it was important to give the user a feeling of acceleration and handling. Applying a different amount of traction was important for game balancing and the fun factor, so that once a player got a feel for the physics, they'd get a sense of accomplishment and become a better racer.

Once again we called on Paper.js which comes with an extensive set of math utilities. We used some of its methods to move the car along the path, while adjusting the car position and rotation smoothly each frame.

```
var trackOffset = _path.length - (_elapsed % _path.length);
var trackPoint = _path.getPointAt(trackOffset);
var trackAngle = _path.getTangentAt(trackOffset).angle;

// Apply the throttle
_velocity.length += _throttle;

if (!_throttle) {
  // Slow down since the throttle is off
  _velocity.length *= FRICTION;
}

if (_velocity.length > MAXVELOCITY) {
```

```
  _velocity.length = MAXVELOCITY;
}

_velocity.angle = trackAngle;
trackOffset -= _velocity.length;
_elapsed += _velocity.length;

// Find if a lap has been completed
if (trackOffset < 0) {
  while (trackOffset < 0) trackOffset += _path.length;

  trackPoint = _path.getPointAt(trackOffset);
  console.log('LAP COMPLETE!');
}

if (_velocity.length > 0.1) {
  // Render the car if there is actually velocity
  renderCar(trackPoint);
}
```

While we were optimizing car rendering, we found an interesting point. On iOS, the best performance was achieved by applying a `translate3d` transform to the car:

```
_car.style.webkitTransform = 'translate3d('+_position.x+'px, '+_position.y+'px,
0px)rotate('+_rotation+'deg)';
```

On Chrome for Android, the best performance was achieved by calculating the matrix values and applying a matrix transform:

```
var rad = _rotation.rotation * (Math.PI * 2 / 360);
var cos = Math.cos(rad);
var sin = Math.sin(rad);
var a = parseFloat(cos).toFixed(8);
var b = parseFloat(sin).toFixed(8);
var c = parseFloat(-sin).toFixed(8);
var d = a;
_car.style.webkitTransform = 'matrix(' + a + ', ' + b + ', ' + c + ', ' + d + ', ' +
_position.x + ', ' + _position.y + ')';
```

Keeping the Devices Synced

The most important (and difficult) part of development was to make sure the game synced across devices. We thought users could be forgiving if a car occasionally skipped a few frames because of a slow connection but it wouldn't be much fun if your car is jumping around, appearing on multiple screens at once. Resolving this required a ton of trial and error, but we eventually settled on a few tricks which made it work.

Calculating Latency

The starting point for syncing devices is knowing how long it takes for messages to be received from the Compute Engine relay. The tricky part is that the clocks on each device will never be completely in sync. To get around this, we needed to find the difference in time between the device and the server.

To find the time offset between the device and the main server, we send a message with the current device timestamp. The server will then reply with the original timestamp along with the timestamp of the server. We use the response to calculate the actual difference in time.

```
var currentTime = Date.now();
var latency = Math.round((currentTime - e.time) * .5);
var serverTime = e.serverTime;
currentTime -= latency;
var difference = currentTime - serverTime;
```

Doing this once is not enough, as the round trip to the server is not always symmetrical, meaning it may take longer for the response to get to the server than it does for the server to return it. To get around this, we poll the server multiple times, taking the median result. This gets us within 10ms of the actual difference between device and server.

Acceleration/Deceleration

When Player 1 presses or releases the screen, the acceleration event is sent to the server. Once received, the server adds its current timestamp and then passes that data along to every other player.

When an "accelerate on" or "accelerate off" event is received by a device, we are able to use the server offset (calculated above) to find out how long it took for that message to be received. This is useful, because Player 1 may receive the message in 20ms, but Player 2 might take 50ms to receive it. This would result in the car being in two different places because device 1 would start the acceleration sooner.

We can take the time it took to receive the event and convert that into frames. At 60fps, each frame is 16.67ms—so we can add more velocity (acceleration) or friction (deceleration) on of the car to account for the frames it missed.

```
var frames = time / 16.67;
var onScreen = this.isOnScreen() && time < 75;

for (var i = 0; i < frames; i++) {
  if (onScreen) {
    _velocity.length += _throttle * Math.round(frames * .215);
  } else {
    _this.render();
  }
}}
```

In the example above, if Player 1 has the car on its screen and the time it took to receive the message is less than 75ms, it will adjust the car's velocity, speeding it up to make up the difference. If the device is not on screen or the message took too long, it will run the render function and actually make the car jump to where it needs to be.

Keeping the Cars Synced

Even after accounting for latency in acceleration, the car could still come out of sync and appear on multiple screens at once; specifically when transitioning from one device to the next. In order to prevent this, update events are sent frequently to keep the cars in the same position on the track across all screens.

The logic is that, every 4 frames, if the car is visible on screen, that device sends its values to each of the other devices. If the car is not visible, the app updates the values with those received and then moves the car forward based on the time it took to get the update event.

```
this.getValues = function () {
  _values.p = _position.clone();
  _values.r = _rotation;
  _values.e = _elapsed;
  _values.v = _velocity.length;
  _values.pos = _this.position;

  return _values;
}
```

```
this.setValues = function (val, time) {
  _position.x = val.p.x;
  _position.y = val.p.y;
  _rotation = val.r;
  _elapsed = val.e;
  _velocity.length = val.v;

  var frames = time / 16.67;

  for (var i = 0; i < frames; i++) {
    _this.render();
  }
}
```

Conclusion

As soon as we heard the concept for Racer, we knew it had the potential to be a very special project. We quickly built a prototype which gave us a rough idea of how to overcome latency and network performance. It was a challenging project that kept us busy during late nights and long weekends, but it was a great feeling when the game started to take shape. Ultimately, we're very happy with the end result. Google Creative Lab's concept pushed the limits of browser technology in a fun way, and as developers we couldn't ask for more.

□□□ #3
□ □□□ □□□ 2026-10-31 22:13:03 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:55 UTC