Rendering performance

From the ground up

HolyJS

$ whoami

Head of Engineering @ Archilogic

HolyJS

Всем привет!

Head of Engineering @ Archilogic

@g33konaut

HolyJS

Всем привет!

Head of Engineering @ Archilogic

@g33konaut

spaces.archilogic.com

HolyJS

What we will look at

```
<html>
  <body>
    <h1>Hello HolyJS</h1>
    <p>Lorem ipsum...</p>
  </body>
</html>
```

Loading a website

1. HTML is fetched from the network

Loading a website

1. HTML is fetched from the network
2. HTML text is parsed into tokens *as it arrives*

Loading a website

1. HTML is fetched from the network
2. HTML text is parsed into tokens *as it arrives*
3. Tokens are parsed into **objects** (DOM / CSSOM)

Loading a website

1. HTML is fetched from the network
2. HTML text is parsed into tokens *as it arrives*
3. Tokens are parsed into **objects** (DOM / CSSOM)
4. Objects are laid out on the page

Loading a website

1. HTML is fetched from the network
2. HTML text is parsed into tokens *as it arrives*
3. Tokens are parsed into **objects** (DOM / CSSOM)
4. Objects are laid out on the page
5. Objects are being painted & composited

Loading a website

1. HTML is fetched from the network
2. HTML text is parsed into tokens *as it arrives*
3. Tokens are parsed into **objects** (DOM / CSSOM)
4. Objects are laid out on the page
5. Objects are being painted & composited
6. JS or CSS can change the content

(2) & (3): Parsing

Parsing

```
<h1>Hello HolyJS!</h1>
```

- H1 element
- Text node

Parsing

```
<h1>Hello HolyJS!</h1>

<p>Lorem ipsum...</p>
```

- H1 element
- Text node
- Paragraph element
- Text node

Parsing

```
<h1>Hello HolyJS!</h1>

<p>Lorem ipsum...</p>

<p><img ...></p>
```

- H1 element
- Text node
- Paragraph element
- Text node
- Paragraph element
- Image element

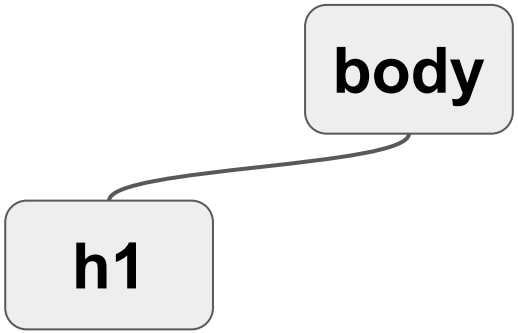
DOM Tree building

- H1 element
- Text node
- Paragraph element
- Text node
- Paragraph element
- Image element

body

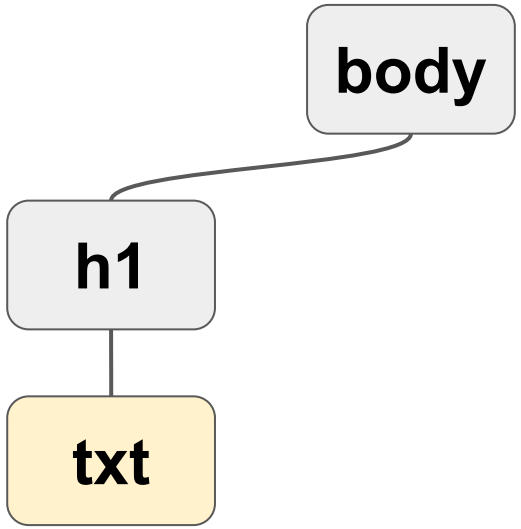
DOM Tree building

- **H1 element**
- Text node
- Paragraph element
- Text node
- Paragraph element
- Image element



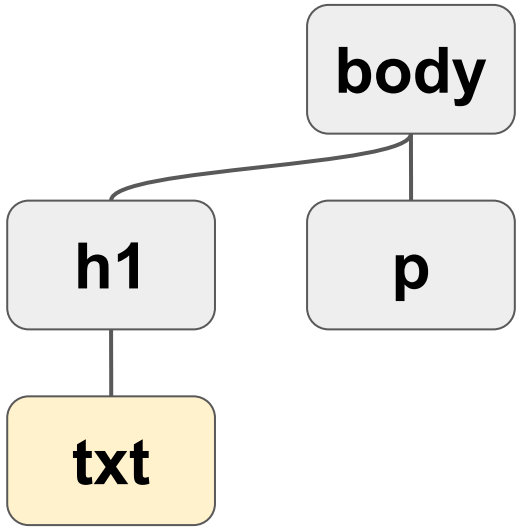
DOM Tree building

- H1 element
- **Text node**
- Paragraph element
- Text node
- Paragraph element
- Image element



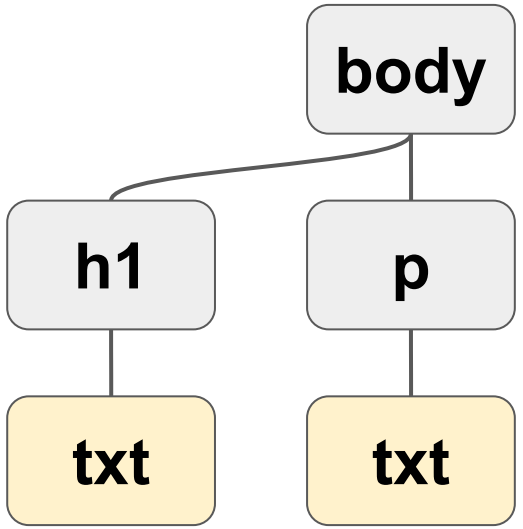
DOM Tree building

- H1 element
- Text node
- **Paragraph element**
- Text node
- Paragraph element
- Image element



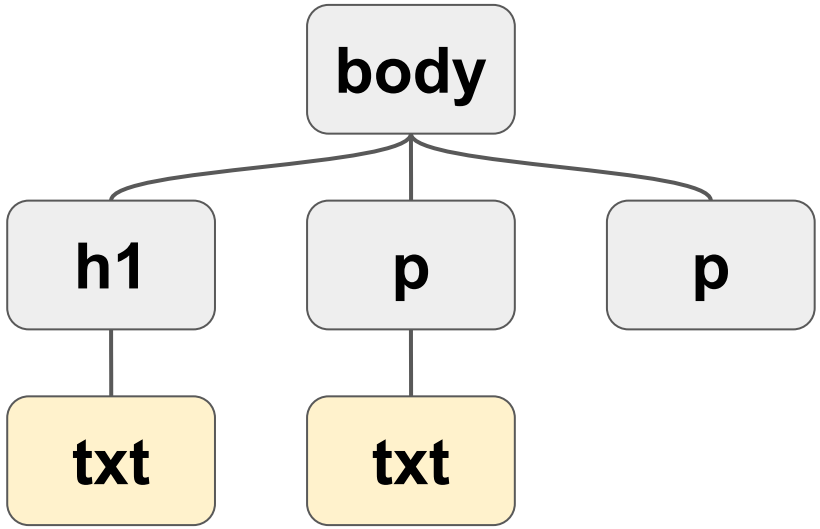
DOM Tree building

- H1 element
- Text node
- Paragraph element
- **Text node**
- Paragraph element
- Image element



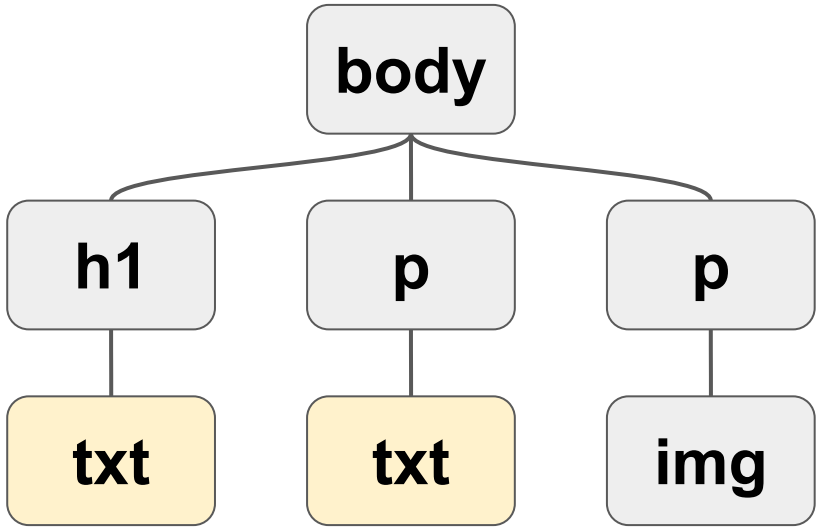
DOM Tree building

- H1 element
- Text node
- Paragraph element
- Text node
- **Paragraph element**
- Image element



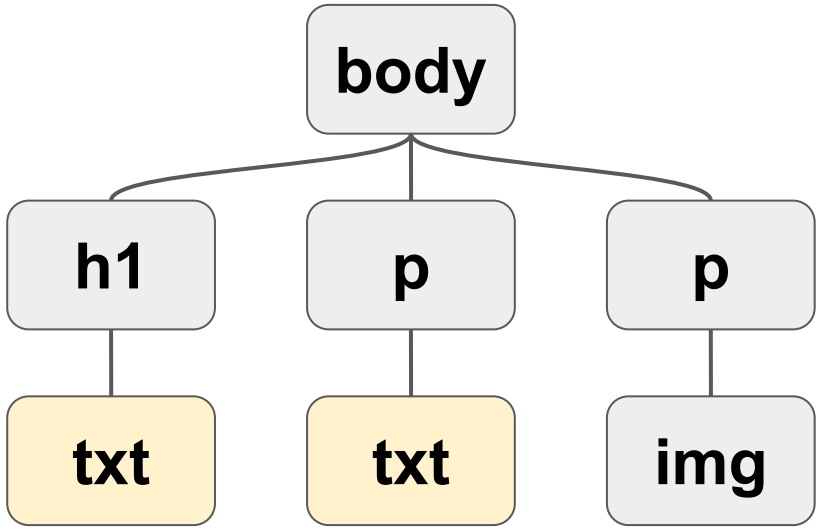
DOM Tree building

- H1 element
- Text node
- Paragraph element
- Text node
- Paragraph element
- **Image element**



DOM Tree building

- H1 element
- Text node
- Paragraph element
- Text node
- Paragraph element
- Image element

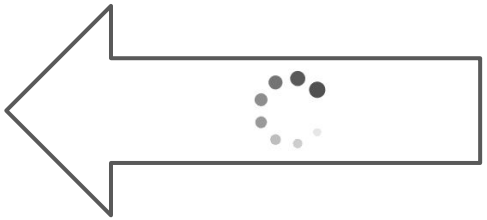


CSSOM building

```
<link rel=”stylesheet” href=”style.css”>

<style>

   body { color: red; }

   h1 { color: blue; }

</style>

<p style=”color: green”>...</p>
```

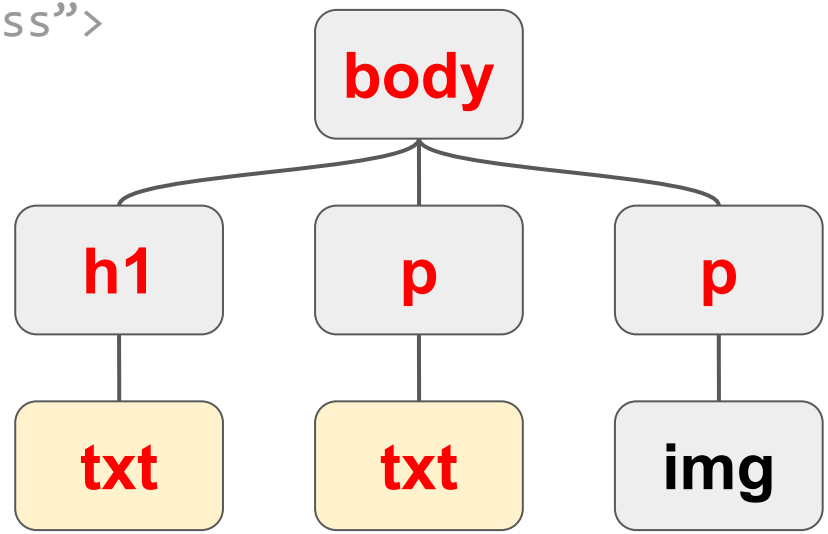
CSSOM building

```
<link rel=”stylesheet” href=”style.css”>
<style>
   body { color: red; }
   h1 { color: blue; }
</style>
<p style=”color: green”>...</p>
```



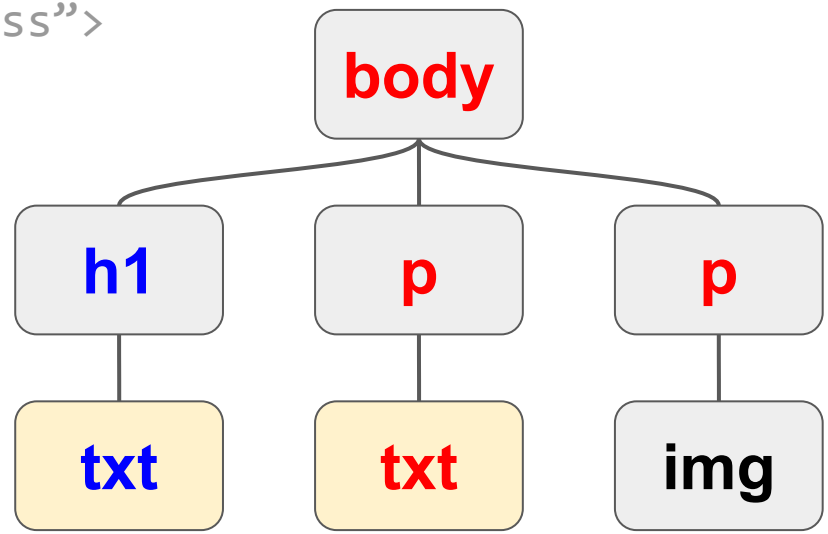
CSSOM building

```
<link rel=”stylesheet” href=”style.css”>
<style>
   body { color: red; }
   h1 { color: blue; }
</style>
<p style=”color: green”>...</p>
```



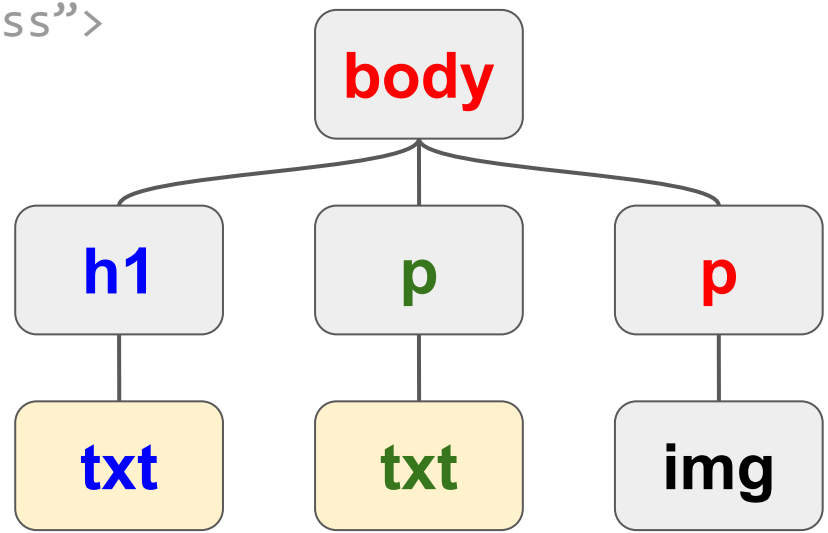
CSSOM building

```
<link rel=”stylesheet” href=”style.css”>
<style>
   body { color: red; }
   h1 { color: blue; }
</style>
<p style=”color: green”>...</p>
```

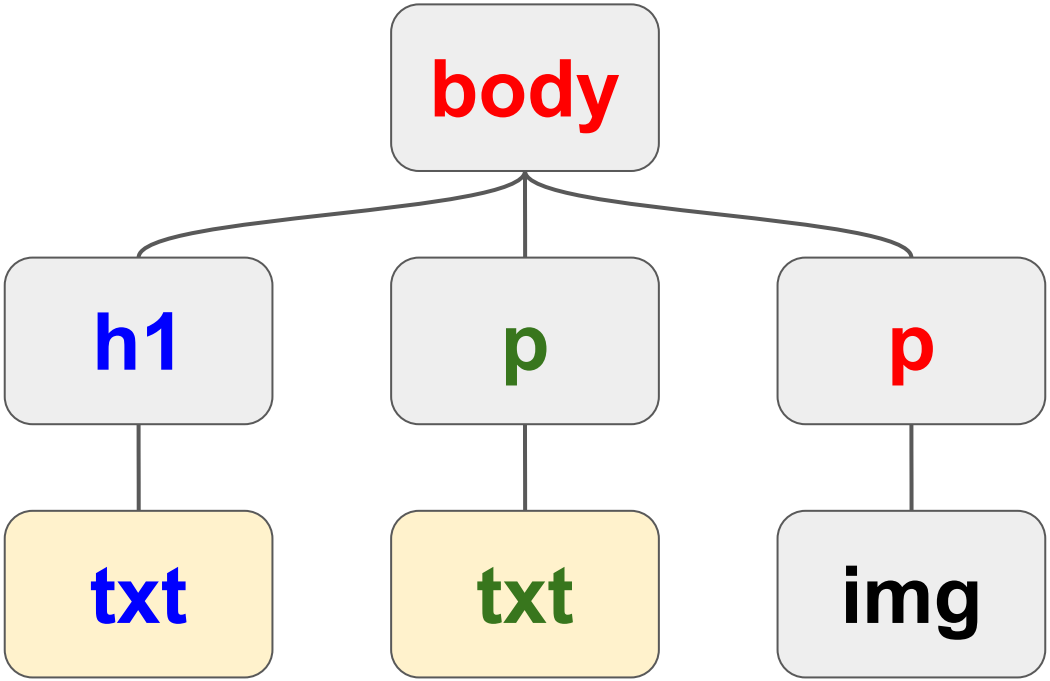


CSSOM building

```
<link rel="stylesheet" href="style.css">
<style>
   body { color: red; }
   h1 { color: blue; }
</style>
<p style="color: green">...</p>
```



CSSOM + DOM



Lessons to learn

- Inline *critical* CSS

Lessons to learn

- Inline *critical* CSS
- HTML source order matters during parsing

Lessons to learn

- Inline *critical* CSS
- HTML source order matters during parsing
- Minimise forcing repeated tree building

Lessons to learn

- Inline *critical* CSS
- HTML source order matters during parsing
- Minimise forcing repeated tree building
- Static HTML for initial render 

(4) Layout process

Layouting

- Where do things go & how large are they?

Layouting

- Where do things go & how large are they?
- Based on CSSOM + DOM

Layouting

- Where do things go & how large are they?
- Based on CSSOM + DOM
- Determines the actual size of each element

Layouting

- Where do things go & how large are they?
- Based on CSSOM + DOM
- Determines the actual size of each element
- Expensive algorithm

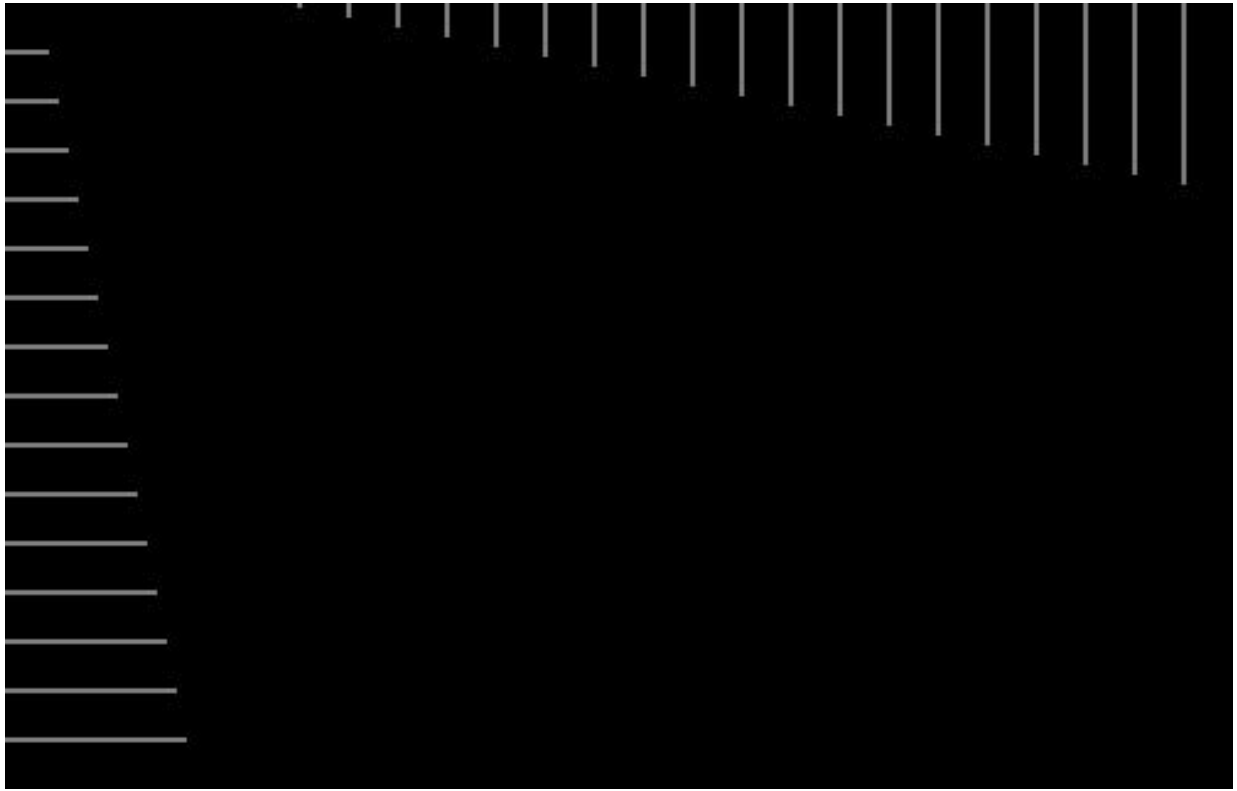
Layouting

- Where do things go & how large are they?
- Based on CSSOM + DOM
- Determines the actual size of each element
- Expensive algorithm
- Change may require repaint

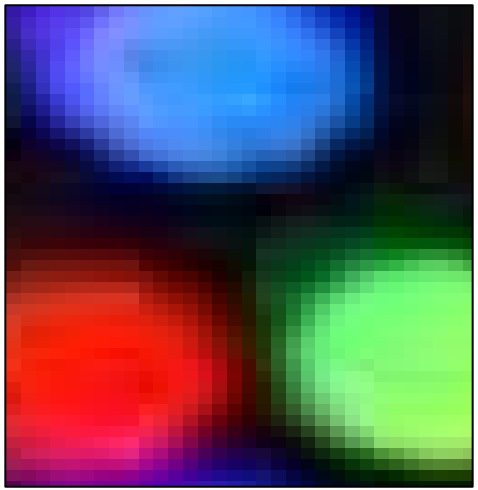
Layouting

- Where do things go & how large are they?
- Based on CSSOM + DOM
- Determines the actual size of each element
- Expensive algorithm
- Change may require repaint
- Animations may cause relayout

5. Painting pixels



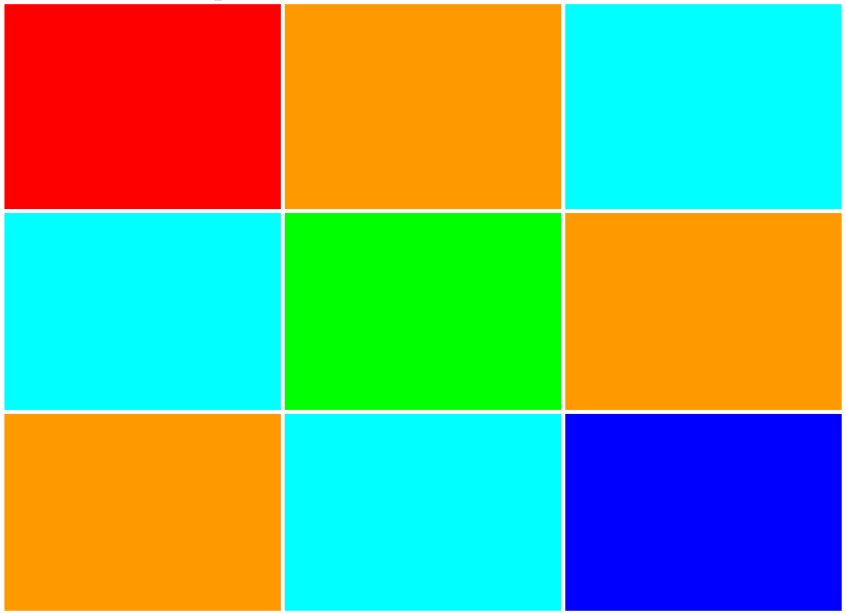
Let's paint



1 pixel = (red, green, blue)

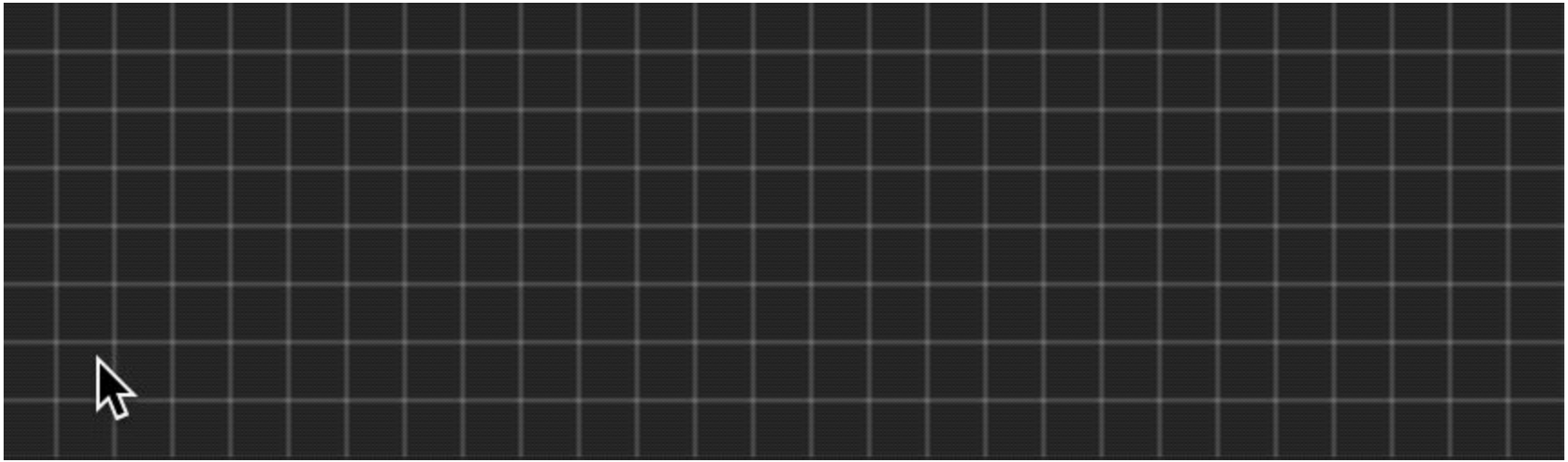
Let's paint

3x3 pixels screen = 3*3*3 values



255 0 0	255 255 0	0 255 255
0 255 255	0 255 0	255 255 0
255 255 0	0 255 255	0 0 255

Painting is...



Painting is...

- Not happening sequentially ;-)

Painting is...

- Not happening sequentially ;-)
- Still expensive...

Painting is...

- Not happening sequentially ;-)
- Still expensive…
- Basically writing into memory

Painting is...

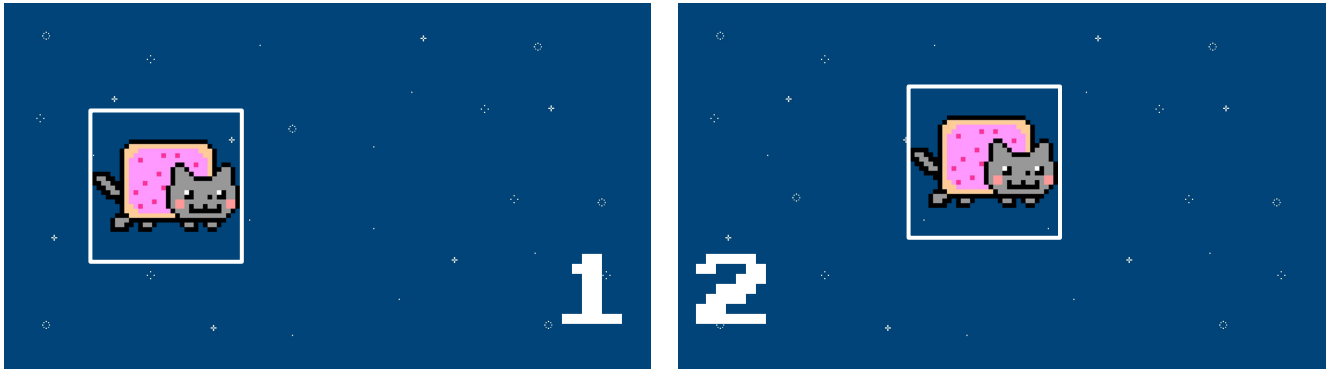
- Not happening sequentially ;-)
- Still expensive…
- Basically writing into memory
- e.g. 500x500 pixel * 3 bytes = 750 kb to write

Compositing

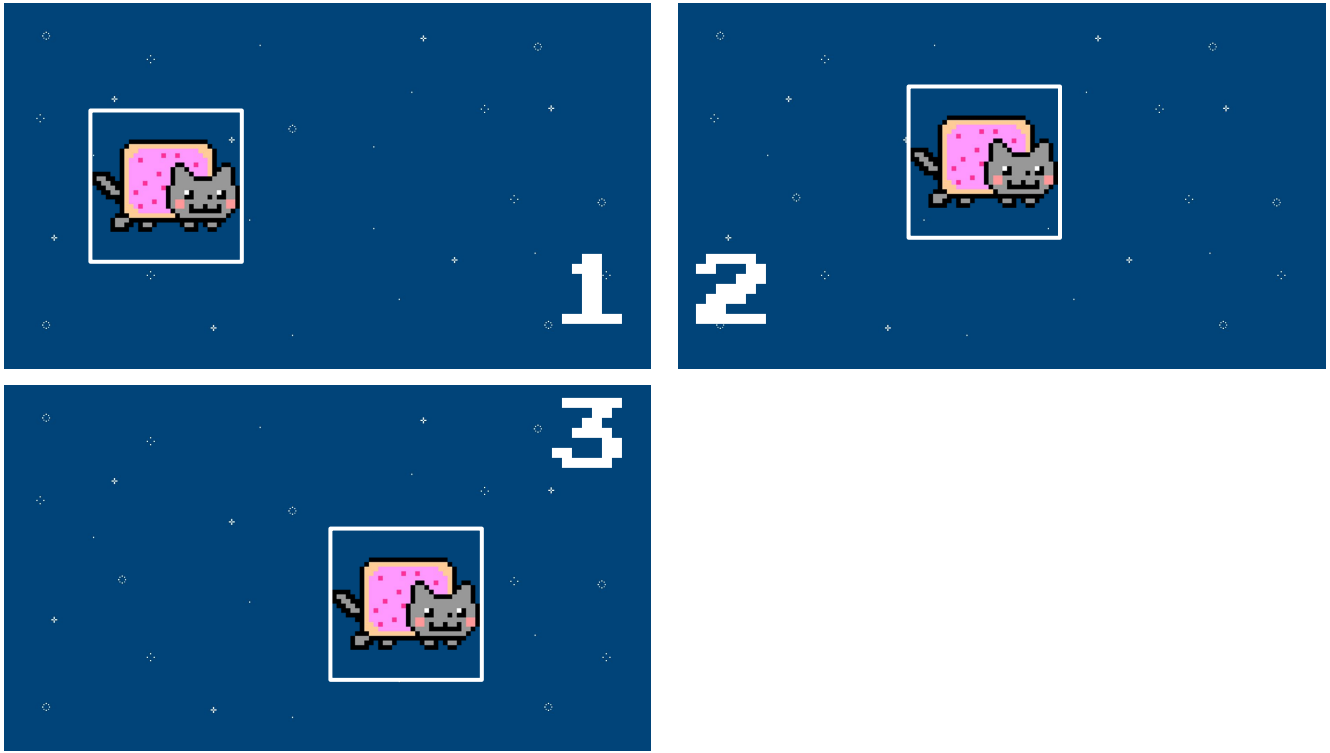
+

Compositing

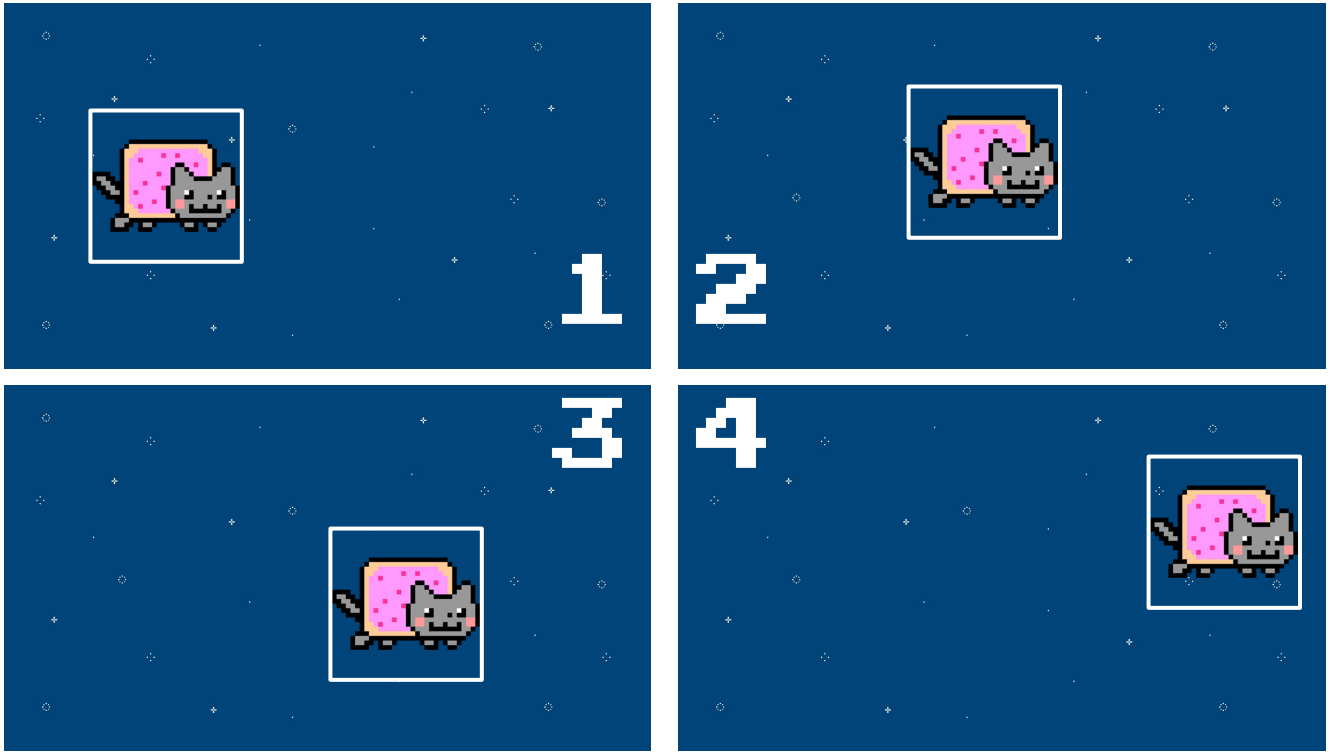
Compositing



Compositing

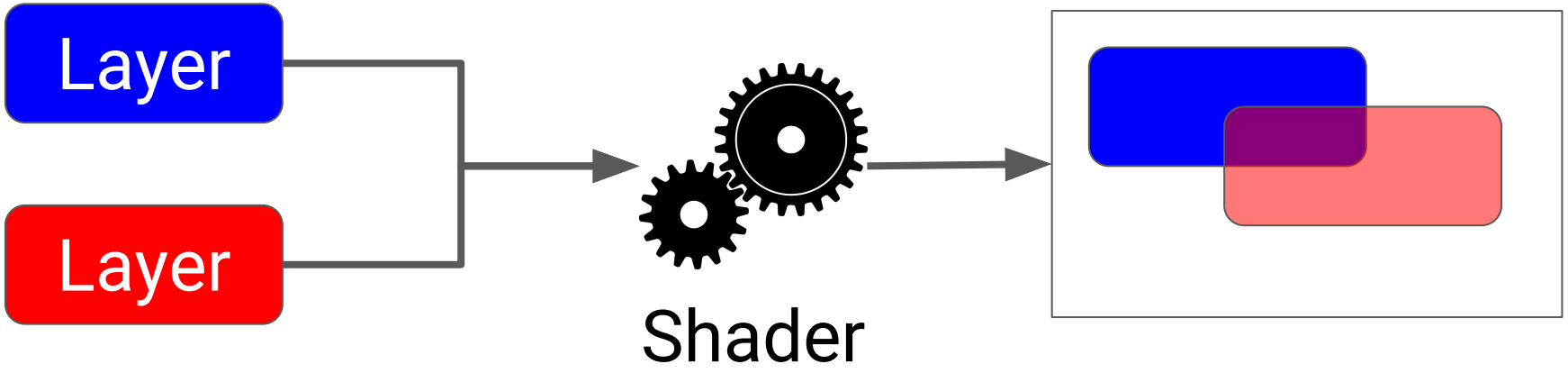


Compositing



Compositing in detail

Compositing in detail



Shader?

```
let shader = (x, y, layers, blend, filter) => {
  return filter(
    blend(x, y, layers) // returns colour
  ) // returns final colour
}
```

Blending?

- Takes a pixel of all layers and combines them

Blending?

- Takes a pixel of all layers and combines them
- Different functions available

Blending?

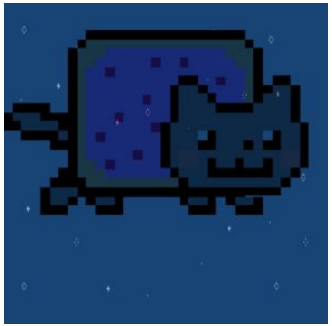
- Takes a pixel of all layers and combines them
- Different functions available
- Example: “**screen**” blending:

```
color(x,y) = 1 - (1 - cat(x,y)) * (1 - sky(x,y))
```

Blending?

- Takes a pixel of all layers and combines them
- Different functions available
- Example: “**multiply**” blending:

```
color(x,y) = cat(x,y) * sky(x,y)
```



Filter?

- Runs a function on the blended colour

Filter?

- Runs a function on the blended colour
- A bunch of filter functions are available

Filter?

- Runs a function on the blended colour
- A bunch of filter functions are available
  - grayscale
  - blur
  - contrast
  - hue-rotate
  - invert
  - opacity
  - saturate
  - sepia
  - drop-shadow

Filter?

- Runs a function on the blended colour
- A bunch of filter functions are available
  - grayscale
  - blur
  - contrast
  - hue-rotate
  - invert
  - opacity
  - saturate
  - sepia
  - drop-shadow
- grayscale: `(r, g, b) => (r + g + b) / 3`

A real shader

```
varying highp vec2 coord;
uniform sampler2D layer;

void main(void) {
  vec4 color = texture2D(layer, vec2(coord.s, coord.t));
  float grayScale = (color.r + color.g + color.b) / 3.0;
  gl_FragColor = vec4(grayScale, grayScale, grayScale, 1.0);
}
```

Compositing

- Keeping individual images separate (*layers*)

Compositing

- Keeping individual images separate (*layers*)
- Fast operation (memory copy)

Compositing

- Keeping individual images separate (*layers*)
- Fast operation (memory copy)
- Can do some transformations without paint

Compositing

- Keeping individual images separate (*layers*)
- Fast operation (memory copy)
- Can do some transformations without paint
  - translate (move)

Compositing

- Keeping individual images separate (*layers*)
- Fast operation (memory copy)
- Can do some transformations without paint
  - translate (move)
  - scale

Compositing

- Keeping individual images separate (*layers*)
- Fast operation (memory copy)
- Can do some transformations without paint
  - translate (move)
  - scale
  - rotate

Layers...

- Layers are **expensive**

Layers...

- Layers are **expensive**
- The browser will be conservative

Layers...

- Layers are **expensive**
- The browser will be conservative
  - `<video>`, `<canvas>`

Layers...

- Layers are **expensive**
- The browser will be conservative
  - `<video>`, `<canvas>`
  - **3D** transforms (*“translateZ hack”*)

Layers...

- Layers are **expensive**
- The browser will be conservative
  - `<video>`, `<canvas>`
  - **3D** transforms (*“translateZ hack”*)
  - **composite-only** animations

Layers...

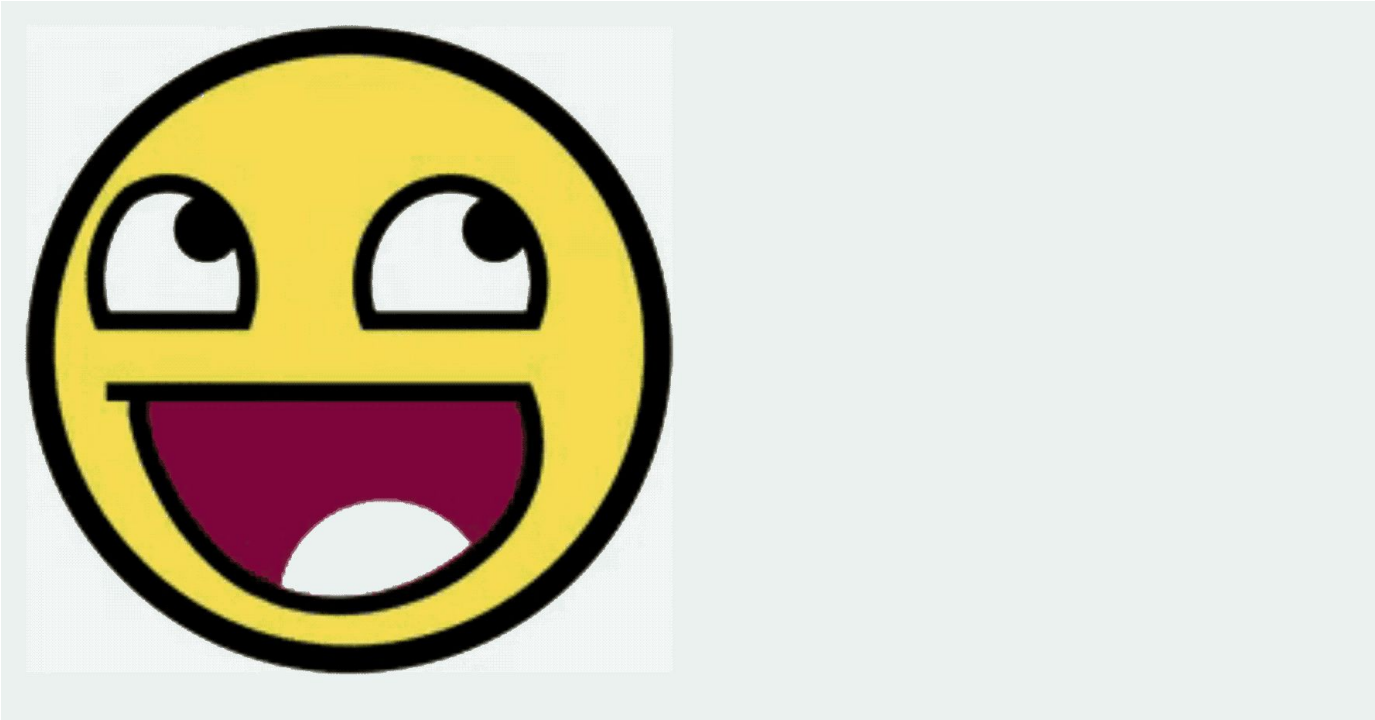
- Layers are **expensive**
- The browser will be conservative
  - `<video>`, `<canvas>`
  - **3D** transforms (*“translateZ hack”*)
  - **composite-only** animations
  - `will-change`

Let's play a game...

```
#transform {
  transform: translateX(150px);
}
setTimeout(() => {
  el.style.transform = 'translateX(0)'
}, 2000)
```

Will it paint?

Yes.



Round 2:

```
#transform { transform: translate3d(150px, 0,0); }


setTimeout(() => {
  el.style.transform = 'translate3d(0, 0, 0)'
}, 2000)
```

Will it paint?

3D transform = layer!

Round 3:

```
@keyframes move {
  0% { left: 0; }
  100% { left: 200px; } }

#transform {
  will-change: left;
  animation: move 2s infinite; }
```

Will it paint?

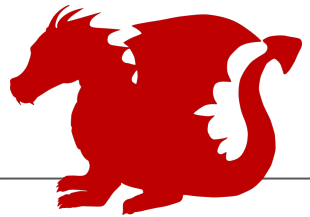
will-change = layer

Elements Cons

will-change.html ×

```
12
13    @keyframes slide {
14      0% {
15        left: 0px;
16      }
17      50% {
18        left: 300px;
19      }
20      100% {
21        left: 0px;
22      }
23    }
24  </style>
25  <img src="cool-yay.jp
26  <script>
27  </script>
28  </body>
29  </html>
```

Will-change & layout



This **only** works if the element isn't in the layout flow

(e.g. `position: absolute`)

Lessons learned

- Avoid relayout & repaint

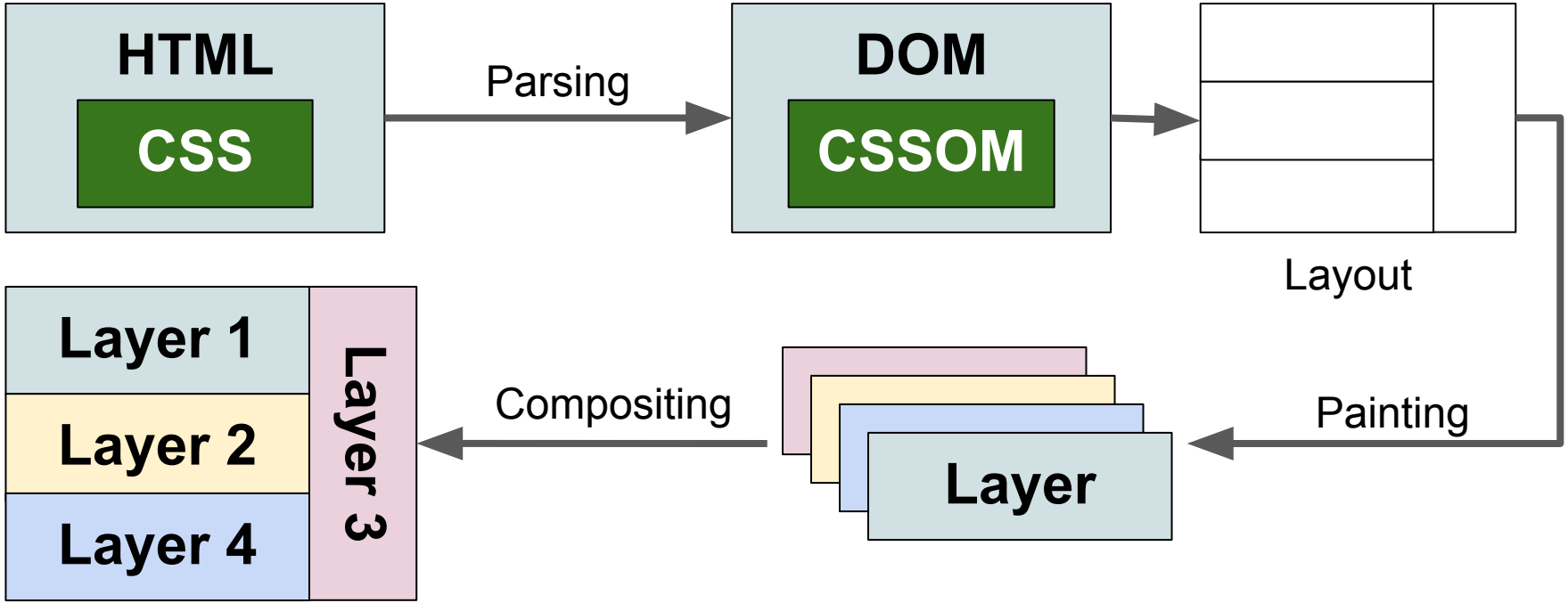
Lessons learned

- Avoid relayout & repaint
- Layers can help, but be careful

Lessons learned

- Avoid relayout & repaint
- Layers can help, but be careful
- 3D transforms are composite-only

Putting it together



Summary

- Make critical content **static & inline**

Summary

- Make critical content **static & inline**
- Minimise **relayout** & **repaint**

Summary

- Make critical content **static & inline**
- Minimise **relayout** & **repaint**
- Use **layers** reasonably

Summary

- Make critical content **static & inline**
- Minimise **relayout** & **repaint**
- Use **layers** reasonably
- **Measure first**, then optimise

“Performance is the *art of avoiding work*”

- Paul Lewis

спасибо

Slides: bit.ly/holyjs17-renderperf

Twitter: @g33konaut

Web: spaces.archilogic.com

HolyJS