# THREE REASONS

three.js

REASON

## OUR MISSION

- Learn some ReasonML
- Build a cool 3D thingy
- Decide if it's Un-ReasonML?

```

'use strict';

var Three = require("./three");
var Three$1 = require("three");
var CrateGif = require("./textures/crate.gif");

function init(element) {
  var rect = element.getBoundingClientRect();
  var scene = new Three$1.Scene();
  var renderer = new Three$1.WebGLRenderer();
  var camera = new Three$1.PerspectiveCamera(70.0, rect.width / rect.height, 1.0, 1000.0);
  camera.position.set(0, 150, 400);
  var texture = new Three$1.TextureLoader().load(CrateGif);
  var material = new Three$1.MeshBasicMaterial({
        map: texture
      });
  var geo = new Three$1.BoxGeometry(125, 125, 125);
  var cube = new Three$1.Mesh(geo, material);
  scene.add(cube);
  renderer.setSize(rect.width, rect.height);
  element.appendChild(renderer.domElement);
  var playing = [/* true */1];
```

## OCaml My Caml

- Rich type system
- FP + OOP
- Cross platform compilation

## Bucklescript & ReasonML

- Compiles to JS
- JS-like syntax
- NPM workflow

## Why should I care?

- Real type safety for your JavaScript
- Excellent foreign function interface
- React ready

## Please Bear With Us...

```
type state = {
  initialized: bool,
  playing: bool
};
type action =
  | Toggle;
let component = ReasonReact.reducerComponent("App");
let make = (_children) => {
  ...component,
  initialState: () => { initialized: false, playing: false },
  reducer: (action, state) =>
    switch action {
      | Toggle => ReasonReact.Update({ ...state, playing: !state.playing })
    },
  render: (self) => {
    let buttonText = self.state.playing ? "Pause" : "Play";
    <div className="App">
      <button onClick={_ => self.send(Toggle)}> (ReasonReact.stringToElement(buttonText)) </button>
    </div>;
  }
};
```

## Working with the DOM

- Using React refs
- Handling nullable data
- Styling in reason-react

## Actions: before

```
type action =
  | Toggle;
```

## Actions: after

```
type action =
  | Ready(option(Dom.element))
  | Toggle;
```

## No Null References!

## Reducer: before

```
reducer: (action, state) =>
  switch action {
    | Toggle => ReasonReact.Update({ ...state, playing: !state.playing })
  },
```

## Reducer: after

```
reducer: (action, state) =>
  switch action {
    | Toggle => ReasonReact.Update({ ...state, playing: !state.playing })
    | Ready(canvas) => ReasonReact.SideEffects(_ => {
      switch (canvas) {
        | (Some(c)) => Js.log(c)
        | _ => ()
      }
    })
  },
```

## Render: before

```
render: (self) => {
  let buttonText = self.state.playing ? "Pause" : "Play";
  <div className="App">
    <button onClick={_ => self.send(Toggle)}> (ReasonReact.stringToElement(buttonText)) </button>
  </div>;
}
```

## Render: after

```
render: (self) => {
  let buttonText = self.state.playing ? "Pause" : "Play";
  <div className="App">
    <button onClick={_ => self.send(Toggle)}> (ReasonReact.stringToElement(buttonText)) </button>
    <div ref={c => self.send(Ready(Js.Nullable.toOption(c)))} />
  </div>;
}
```

```
type state = {
  initialized: bool,
  playing: bool
};
type action =
  | Ready(option(Dom.element))
  | Toggle;
let component = ReasonReact.reducerComponent("App");
let make = (_children) => {
  ...component,
  initialState: () => { initialized: false, playing: false },
  reducer: (action, state) =>
    switch action {
      | Toggle => ReasonReact.Update({ ...state, playing: !state.playing })
      | Ready(canvas) => ReasonReact.SideEffects(_ => {
        switch (canvas) {
          | (Some(c)) => Js.log(c)
          | _ => ()
        }
      })
    },
  render: (self) => {
    let buttonText = self.state.playing ? "Pause" : "Play";
    <div className="App">
      <button onClick={_ => self.send(Toggle)}> (ReasonReact.stringToElement(buttonText)) </button>
      <div ref={c => self.send(Ready(Js.Nullable.toOption(c)))} />
    </div>;
  }
};
```

## CSS in JS?

**Sorry, not yet.**

## How about inline styles?

Ok

## Optional labeled arguments

```
let canvasStyle = ReactDOMRe.Style.make(~height="100vh", ());
```

```
type controller = { playPause: unit => unit };
let init = _ => {
  let playing = ref(true);
  {
    playPause: () => {
      let nextState = !(playing^);
      Js.log(nextState ? "Playing" : "Paused");
      playing := nextState;
    }
  }
};
```

## State: before

```
type state = {
  initialized: bool,
  playing: bool
};
```

## State: after

```
open Game;
type state = {
  initialized: bool,
  playing: bool,
  controller: option(controller)
};
```

## Actions: before

```
type action =
  | Ready(option(Dom.element))
  | Toggle;
```

## Actions: after

```
type action =
  | Start(controller)
  | Ready(option(Dom.element))
  | Toggle;
```

## Reducer: before

```
reducer: (action, state) =>
  switch action {
    | Toggle => ReasonReact.Update({ ...state, playing: !state.playing })
    | Ready(canvas) => ReasonReact.SideEffects(_ => {
      switch (canvas) {
        | (Some(c)) => Js.log(c)
        | _ => ()
      }
    })
  },
open Game;
type state = {
  initialized: bool,
  playing: bool,
  controller: option(controller)
};
type action =
  | Start(controller)
  | Ready(option(Dom.element))
  | Toggle;
let component = ReasonReact.reducerComponent("App");
let canvasStyle = ReactDOMRe.Style.make(~height="100vh", ());
let make = (_children) => {
  ...component,
  initialState: () => { initialized: false, playing: false, controller: None },
  reducer: (action, state) =>
    switch action {
      | Start(c) => ReasonReact.Update({ initialized: true, playing: true, controller: Some(c) })
      | Toggle => ReasonReact.UpdateWithSideEffects({ ...state, playing: !state.playing }, self => {
        switch (self.state.controller) {
```

```
type scene;
type renderer;
type domElement;
type childObject;
type camera = childObject;
type geometry;
type material;
type mesh;
type vector;
type loader;
type texture;
type materialSpec = {. "map": texture };
```

## Defined As:

```
[@bs.new] [@bs.module "three"] external boxGeo: (int, int, int) => geometry = "BoxGeometry";
```

## Used like:

```
let geo = boxGeo(125, 125, 125);
```

## Compiles to:

```
var geo = new Three$1.BoxGeometry(125, 125, 125);
```

## Defined as:

```
[@bs.new] [@bs.module "three"] external textureLoader: unit => loader = "TextureLoader";
[@bs.send.pipe : loader] external load : string => texture = "load";
```

## Used like:

```
let texture = textureLoader() |> load("path/to/texture.gif");
```

## Compiles to:

```
var texture = new Three$1.TextureLoader().load("path/to/texture.gif");
```

## Defined as:

```
[@bs.get] external getDomElement : renderer => domElement = "domElement";
```

## Used like:

```
let gameCanvas = renderer |> getDomElement;
```

## Or maybe like this?

```
let gameCanvas = getDomElement(renderer);
```

## Compiles to:

```
var gameCanvas = renderer.domElement;
```

```
/* Our types! */
type scene;
type renderer;
type domElement;
type childObject;
type camera = childObject;
type geometry;
type material;
type mesh;
type vector;
type loader;
type texture;
type materialSpec = {. "map": texture };
/* Constructurs */
[@bs.new] [@bs.module "three"] external newScene: unit => scene = "Scene";
[@bs.new] [@bs.module "three"] external newRenderer: unit => renderer = "WebGLRenderer";
[@bs.new] [@bs.module "three"] external newCamera: (float, float, float, float) => camera = "PerspectiveCamera";
[@bs.new] [@bs.module "three"] external boxGeo: (int, int, int) => geometry = "BoxGeometry";
[@bs.new] [@bs.module "three"] external basicMeshMaterial: materialSpec => material = "MeshBasicMaterial";
```

```

open Three;

[@bs.module] external crate : string = "./textures/crate.gif";

type controller = {
  playPause: unit => unit
};

let init = element => {
  let unwrapped = ReactDOMRe.domElementToObj(element);
  let rect = unwrapped##getBoundingClientRect();

  let scene = newScene();
  let renderer = newRenderer();
  let camera = newCamera(70.0, rect##width /. rect##height, 1.0, 1000.0);
  camera |> setPosition(0, 150, 400);

  let texture = textureLoader() |> load(crate);
  let material = basicMeshMaterial({ "map": texture });
  let geo = boxGeo(125, 125, 125);
  let cube = newMesh(geo, material);

  scene |> add(cube);

```

## THREE REASONS WHY

- First rate type system for JavaScript
- Easy integration with Node projects
- The language is cool AF

## THREE REASONS WHY NOT

- Immaturity
- High velocity
- Added complexity

THANKS!