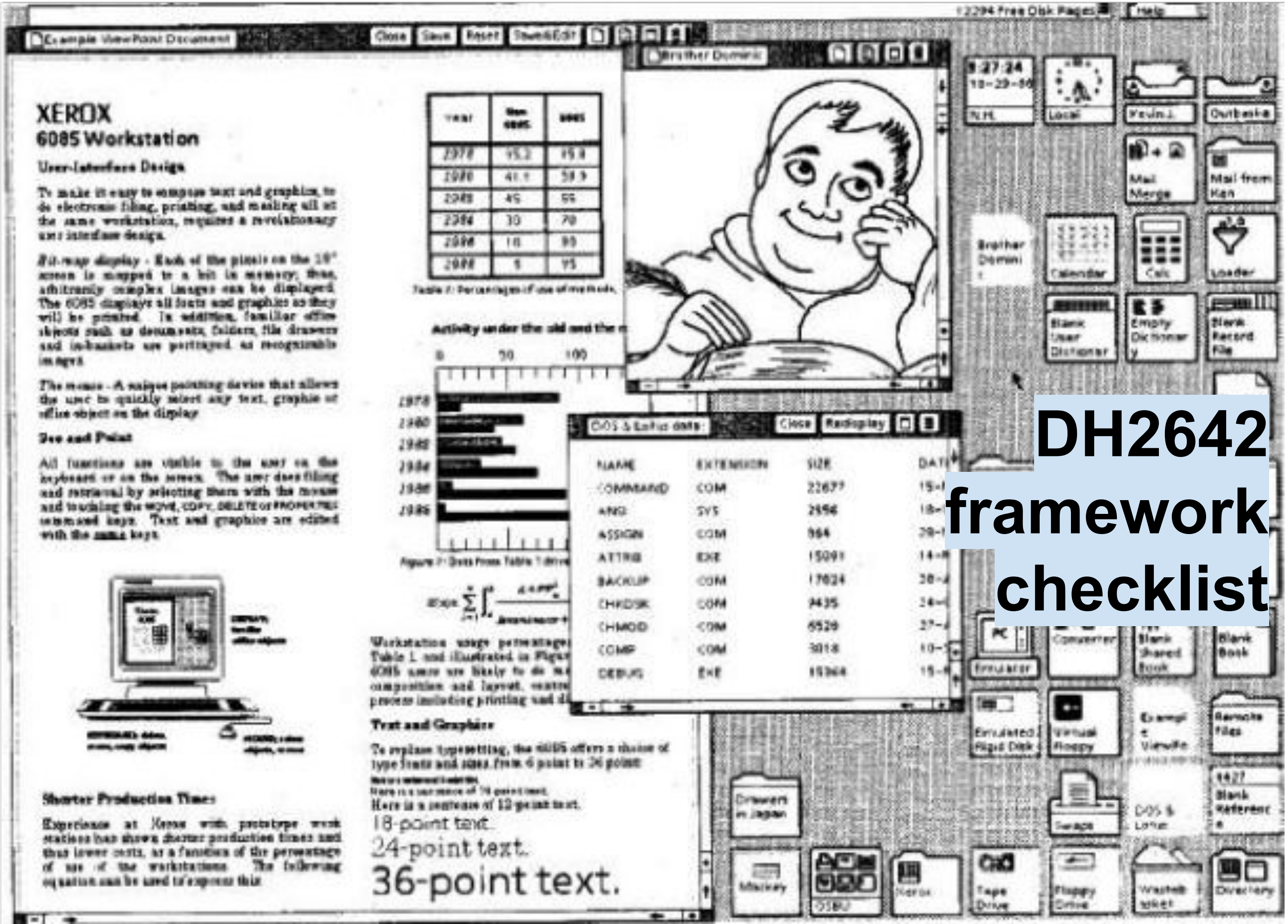
DH2642 framework checklist



XEROX
6085 Workstation

User-Interface Design

To make it easy to compose text and graphics, to do electronic filing, printing, and mailing all at the same workstation, requires a revolutionary user interface design.

Bit-map display - Each of the pixels on the 19" screen is mapped to a bit in memory; thus, arbitrarily complex images can be displayed. The 6085 displays all fonts and graphics as they will be printed. In addition, familiar office objects such as documents, folders, file drawers and in-baskets are portrayed as recognizable images.

The mouse - A unique pointing device that allows the user to quickly select any text, graphic or office object on the display.

See and Point

All functions are visible to the user on the keyboard or on the screen. The user does filing and retrieval by selecting them with the mouse and touching the MOVE, COPY, DELETE or PROPERTIES command keys. Text and graphics are edited with the same keys.

Year	Non-6085	6085
1978	15.2	15.8
1980	41.1	58.9
1982	45	55
1984	30	70
1986	10	90
1988	5	95

Table 1: Percentages of use of methods.

Activity under the old and the n

Figure 1: Data from Table 1 drive

Workstation usage percentage Table 1 and illustrated in Figur 6085 users are likely to do m composition and layout, contr process including printing and d

Text and Graphics

To replace typesetting, the 6085 offers a choice of type fonts and sizes from 6 point to 36 point.

Here is a sentence of 10 point text.
Here is a sentence of 12 point text.
18-point text.
24-point text.
36-point text.

Shorter Production Times

Experience at Xerox with prototype work stations has shown shorter production times and thus lower costs, as a function of the percentage of use of the workstations. The following equation can be used to express this

DOS & Lotus data:

NAME	EXTENSION	SIZE	DATE
COMMAND	COM	22677	15-
ANSI	SYS	2956	18-
ASSIGN	COM	964	20-
ATTRIB	EXE	15091	14-
BACKUP	COM	17024	28-
CHKDSK	COM	9435	24-
CHMOD	COM	6529	27-
COMP	COM	3018	10-
DEBUG	EXE	15364	15-

Framework checklist

Layout (templating? Binding? HTML familiarity)

- Populating View with Model data
- Array data
- Conditional View population

Interaction

- Two-way-binding?
- Controller separation?

Components

- Inputs = Props
- Outputs: new events, but no bubbling
- Lifecycle

Model-ViewModel connection / injection

Navigation (Router) ?

State management ?

Remote data support?

Checklist:

- map MVC and other concepts to the framework
- Compare frameworks
- Relate new concepts introduced by framework to MVC
- Use the checklist when new frameworks and platforms come about

Vue “model” and mounting on DOM

```
new Vue({
   data: {
     shapes: [    { type: 'ellipse', x: 10, y: 10, w: 12, h: 34 },...      ],
     currentShape:0,
     shapeType:'rectangle',
     drawing:false
   },
 }).$mount("#app");
```

data *is more of a ViewModel*

Vue checklist

Layout (templating? Binding? HTML familiarity)

Populating View with Model data {{expr}}   <tag v-bind:attribute=”expr” ..>   :attribute=”expr”

Array data <tag v-for=”elem in array” ..>   v-for=”(elem, index) in array”   key

Model-View separation?

```
<tr v-for="dish in menu" :key="id">
      <td>{{dish.name}}</td>
</tr>
```

{{expression}} is called ***interpolation*** *or* ***moustache***

:attribute is shorthand for v-bind:attribute
v-bind**:key** recommended for v-for to help Vue distinguish the elements

Are Model and View still separated in Vue?

Does the View know too much about the model internals?

How could the model protect its business better?

Vue checklist

Layout

Populating View with Model data {{expr}} <tag v-bind:attribute=”expr” ..>

Array data <tag v-for=”elem in array” ..> v-for=”(elem, index) in array” key

Conditional View population <tag v-if=”expr” ..> v-else v-else-if

Interaction <tag v-on:click=”doSomething()” ..> @click=

Two-way-binding? <tag v-model=”expr” >

equivalent with <tag v-bind:value=”expr” v-on:input=”$emit(‘input’, $event.target.value)” >

Interaction, events, v-on

```
<button v-on:click="menu.push(dish)">Add to menu</button>
```

v-on:click shorthand: **@click**

push() and other array methods are **reactive.** Vue will recognize them and update the bindings.

- array[index]=value in Javascript is not reactive. Use **Vue.set(array, index, value)**
- v-on:click=”array[index]=value” will work because it’s parsed by Vue first

Are Model and Controller still separated in Vue?

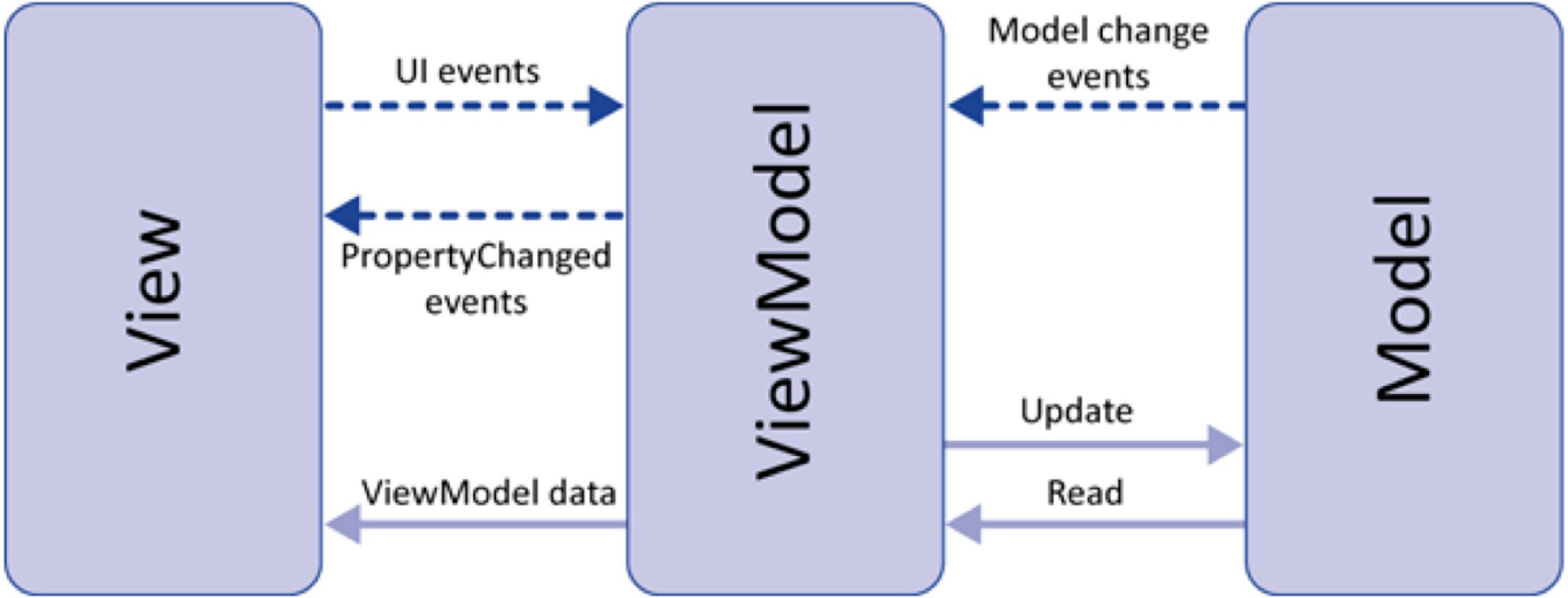
Does the Controller know too much about the model internals?

View-Controller separation in Vue

```
<button v-on:click="menu.push(dish.id)">Add to menu</button>
```

Are View and Controller separated in Vue.js?

How could we improve the separation?



View

UI events

PropertyChanged events

ViewModel data

ViewModel

Model change events

Update

Read

Model

Components

Components are a way to write own views and access them like HTML elements

HTML

- **Standard tag name**, e.g. <input ..>
- **Attributes**, e.g. type=...
- **Events**   e.g. onchange=”...”
- **State** (e.g. has keyboard focus)

The typical app in all of Vue, React, Angular, will be a hierarchy of custom components

Component

- **Custom name**, e.g. <MyComponent ..>
- **Props**, e.g. shapes=...
- **Custom events**   e.g. addShape=”...”
- **State** if needed

Custom events do not bubble! They are only available to the parent component
Props down, events up

Vue checklist (Canvas page)

Layout (templating? Binding? HTML familiarity)

- Populating View with Model data {{expr}} <tag v-bind:attribute=”expr” ..>
- Array data <tag v-for=”elem in array” ..> v-for=”(elem, index) in array” key
- Conditional View population <tag v-if=”expr” ..> v-else v-else-if

Interaction <tag v-on:click=”doSomething()” ..>

- Two-way-binding? <tag v-model=”expr” >
  - equivalent with <tag v-bind:value=”expr” v-on:input=”$emit(‘input’, $event.target.value)” >
- Controller separation? Not directly

Components Vue.component(‘selector’, { template:’<tag>...’, props:[‘prop1’,...] })

- Inputs props
- Outputs: new events $emit
- Lifecycle created(), mounted(), updated(), destroyed()

```
Vue.component("shape-type", {
 template: `<select v-bind:value="value"
                  v-on:input="$emit('input', $event.target.value)">
        <option>rectangle</option>
       <option>line</option>
       <option>ellipse</option>
         </select>`       ,
   props: [ "value" ]
 });
```

Template Vue component written in JavaScript, the template is an ordinary JavaScript string…
Not a DOM template so no syntax highlighting...

*This component is **stateless**. Components that want to keep state must declare a **data**() function, along with **template** and **props***

```
<shape-type v-bind:value=”someVar” v-on:input=”someVar=$event” />
<shape-type :value=”someVar” @input=”someVar=$event” />

 <shape-type v-model=”someVar” />
```

Vue single-file component (.vue extension)

```
<template>
 <div>
   <input :value="value" @input="$emit('update', $event.target.value)">
 </div>
</template>
<script>
export default {
 props: ["value"]
};
</script>
<style>
input {
 width: 2em;
}</style>
```

Keeps the component separate and can do styling
Syntax highlighting like in DOM templates

It is transpiled to JavaScript by Vue tools

Vue render() function, with hyperscript or JSX

```
export default {
 props: ["currentShape", "shapes", "changedCurrent"],
 render() {
   return (
     <select value={this.currentShape} oninput={e => this.changedCurrent(e.target.value)} >
       {this.shapes.map((shape, i) => (
         <option value={i}>
           {shape.type} {shape.x} {shape.y} {shape.w} {shape.h}
         </option>
       ))}
     </select>
   );
 }
};
```

*This component does not emit an event, but simply calls a callback (**changedCurrent**) passed by the parent. React uses the same technique.*

*Cannot use v-on:input but passes a callback to **oninput**. React uses the same technique.*

*Data is still **reactive** here, but JS limitations apply. Use Vue.set(...)*

See also the JSX presentation

Red curly braces {} indicate transition between JSX "tags" and JavaScript

*Vue hyperscript function is called **Vue.createElement** and is sent as render() parameter*

Vue checklist (Canvas)

Layout (templating? Binding? HTML familiarity)

Populating View with Model data {{expr}} <tag v-bind:attribute=”expr” ..>

Array data <tag v-for=”elem in array” ..> v-for=”(elem, index) in array” key

Conditional View population <tag v-if=”expr” ..> v-else v-else-if

Interaction <tag v-on:click=”doSomething()” ..>

Two-way-binding? <tag v-model=”expr” >

equivalent with <tag v-bind:value=”expr” v-on:input=”$emit(‘input’, $event.target.value)” >

Controller separation? Not directly

Components Vue.component(‘selector’, { template:’<tag>...’, props:[‘prop1’,...] })

Inputs props

Outputs: new events $emit

Lifecycle created(), mounted(), updated(), destroyed()

Model-ViewModel connection / injection data() of each Component

Navigation (Router) ? `router: new VueRouter([{path: ‘path:id’, component:Component}, ...] )`

State management ? Vuex, Redux,...

Remote data/asynchronous support? use mounted() to do the fetch() and set state (as in React)

A few Vue remarks

The reactivity magic and binding make programming a simple application a breeze

However, when going into details, lots of small exceptions pop up, and a new Vue feature needs to be introduced (see Vue.set(...))

Feature set is not minimalistic. Things can be done in many ways

In contrast, React has less magic and fewer features

Stark competition with React lately (hyperscript, JSX, see “same as in React” on slides)

JSX React and Vue: Who calls the callback?

```
export default {
 props: ["currentShape", "shapes", "changedCurrent"],
 render() {
   return (
     <select value={this.currentShape} oninput={e => this.changedCurrent(e.target.value)} >
       {this.shapes.map((shape, i) => (
         <option value={i}>
           {shape.type} {shape.x} {shape.y} {shape.w} {shape.h}
         </option>
       ))}
     </select>
   );
 }
};
```

Usage:
```
<shape-selector-func
     :shapes="shapes"
     :currentShape="currentShape"
     :changedCurrent="myCallback"
   />
```

Don’t call your callback! Who calls?

Higher-order function (e.g. array.map, array.reduce, array.filter…) :
Programmer writes f(x){... } **array**.filter(f) filter() **calls** f(arrayElement). **f is not a callback because HOF are synchronous!**

Timeout, Interval
todoLater(){...} setTimeout(todoLater, when) setTimeout impl. calls todoLater()

Promise
cb(data){...} promise.then(cb) promise implementation **calls** cb(resolveResult)

Observer
update(){...} , model.addObserver(observer) **model**.notifyObservers() **calls** observer.update(event) Redux
reducer(state, action){...} createStore(reducer) **Redux calls** reducer(state, action) reducer is **not a callback! It’s usually invoked synchronously**

Don’t call your callback! Who calls?

DOM Event

cb(event){...}   elem.addEventListener(‘click’, cb)   **Browser calls** cb(window.event)

Vue DOM events

<input v-on:change=”cb” />   Browser **calls** cb(window.event)

Typically $emit is called when treating a DOM event, or a custom event from a child component

Vue custom events

<MyComponent v-on:customEvent=”cb” />

MyComponent calls $emit(‘customEvent’, object)   Vue **calls** cb(object)

JSX DOM event

<input onchange={cb} />   Browser **calls** cb(window.event)

Typically eventProp(params) is called when treating a DOM event

JSX callback event **(props down, events up)**

<MyComponent eventProp={cb} />

MyComponent calls eventProp(params)   therefore MyComponent **calls** cb(params)

JSX “props down, events up” example (pseudocode)

```
class Parent{
    cb(){...}
    render(){
      <Child  eventProp={cb}  />
    }
}
```

eventProp is sent from Parent to Child (down)

eventProp() is called by child into parent code (up)

props down, events up

```
class Child{
    render(){
          return <input onchange={e=>this.eventProp(e.someField)} />   // therefore calls cb(e.someField)
    }
}
```

JSX “props down, events up” example (pseudocode)

```
class Parent{
    cb(){...}
    render(){
      <Child  eventProp={cb}  />
    }
}
class Child{
    childCb(){...}
    render(){
         Return <GrandChild eventChildProp= {param=>this.eventProp(param)} />    // calls cb(param)
    }
}
```

ShapeSelector, Vue (recap)

```
export default {
 props: ["currentShape", "shapes", "changedCurrent"],
 render() {
   return (
     <select value={this.currentShape} oninput={e => this.changedCurrent(e.target.value)} >
       {this.shapes.map((shape, i) => (
         <option value={i}>
           {shape.type} {shape.x} {shape.y} {shape.w} {shape.h}
         </option>
       ))}
     </select>
   );
 }
};
```

Usage:

```
<shape-selector-func
  :shapes="shapes"
  :currentShape="currentShape"
  :changedCurrent="myCallback"
/>
```

ShapeSelector, React version

Play with it in Stackblitz!

```
class ShapeSelector extends Component {
 constructor(){ super (); }
 render() {
   return (
     <select value={this.props.currentShape}
             onInput={e => this.props.changedCurrent(e.target.value)} >
       {this.props.shapes.map((shape, i) => (
         <option value={i}>
           {shape.type} {shape.x} {shape.y} {shape.w} {shape.h}
         </option>
       ))}
     </select>
   );
 }
};
```

this.props.propName *instead of*

this.propName

onInput *instead of oninput*

Usage: <ShapeSelector

```
    shapes={JS array expr}
    currentShape={JS integer expr }
    changedCurrent={ JS function expr }
  />
```

React State and app DOM mount

Play with it in Stackblitz!

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class App extends Component {
  constructor() {
    super();
    this.state = {
      shapes: [   { type: 'ellipse', x: 10, y: 10, w: 12, h: 34 },    { type: 'line', x: 60, y: 20, w: 12, h: 34 },     { type: 'rectangle', x: 30, y: 30, w: 12, h: 34 }, ],
      currentShape: 0
    };
  }
  render() {    return  <div> <ShapeSelector shapes={this.state.shapes}
                                                currentShape={this.props.currentShape}
                                                changedCurrent={c=>this.setState(state=>{state.currentShape=c; return state;})}   />        </div> ;
  }
}
render(<App />, document.querySelector('#root'));
```

The React app "model" is usually the State of the top-level component. "Push state up"

React functional components and state hook

Play with it in Stackblitz!

```
const ShapeDisplay= ({shapes, currentShape})=>    // remember object destructuring ?
   shapes.map((shape, index)=> <div className={index==currentShape?"currentShape":"shape"}>{shape.type}</div>)


const App=()=>{
  const [currentShape, setCurrentShape]=useState(0);    // array destructuring
  const [shapes]= useState([ { type: 'ellipse', x: 10, y: 10, w: 12, h: 34 }       { type: 'line', x: 60, y: 20, w: 12, h: 34 }      ]);
   return <div>
        <ShapeDisplay shapes={shapes}
            currentShape={currentShape} />


      <ShapeSelector shapes={shapes}
            currentShape={currentShape}
            changedCurrent={c=>setCurrentShape(c)}
            />
            </div>;
}
```

useState(initialValue) is a **state hook**.
Returns a two element array

1. current value
2. value setter callback

Such functional hooks are available for other React functionality for programmers who want to avoid creating class components

React State and Vue data()

They have the same function

State change triggers rendering update. **Props cannot be changed!**

- React:
  - after setState(), sub-components whose properties change are determined
  - render() is called in virtual DOM, then compare to the browser DOM (**reconciliation**)
- Vue
  - Reactive properties or Vue.set() trigger re-render
  - Binding: bound values in templates are updated
  - JSX render() is called, it produces Vue template, not DOM!

State example basic draw app

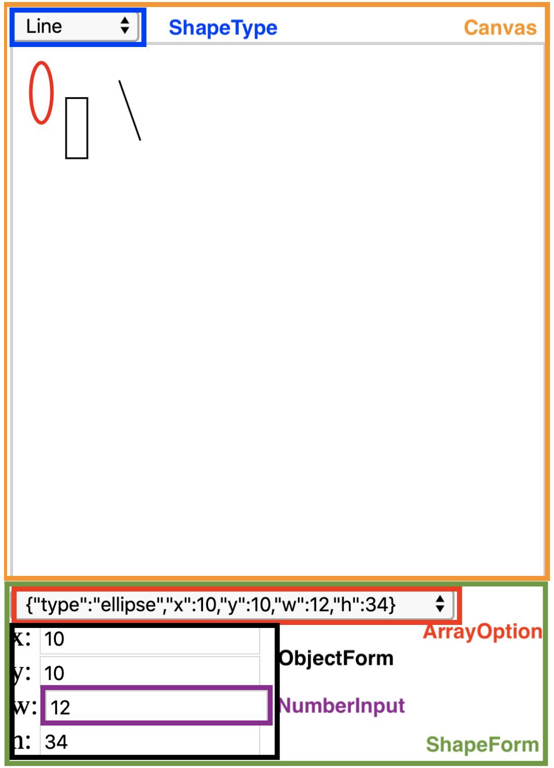
Vue single-component with shapes[], currentShape, drawing, shapeType ("push state up")

Vue component implementation

- App shapes[], currentShape
  - ShapeCanvas: drawing, shapeType
  - ShapeForm: no state

React component implementation(s)

- App with shapes[], currentShape
  - ShapeCanvas with creatingShape, shapeType
  - ShapeForm, no state
    - 4x NumberInput: dirty



Line

{"type":"ellipse","x":10,"y":10,"w":12,"h":34}

x: 10

y: 10

w: 12

h: 34

React (and Vue) remote data

When using remote data, the objective is to trigger re-render when the data has arrived. We do that via setState() and set the fetch() result in the new state

The standard approach is to call the fetch() in **componentDidMount**()

- lifecycle method.
- Others are available: componentDidUpdate(), componentWillUnmount()
- useEffect(callback) hook, used for all three!
- Vue.js correspondent: mounted()

Container and Presentation Components (views)

What's the difference between these two types of components?

Container

- Concerned with *how things work*
- Know about the ~~model~~ **App state**
- Load the data from APIs, pass the data through properties to child components
- sends modifications to the model/state based on events from presentation component
- Include only presentation component(s) and no HTML

Presentation

- Concerned with *look and feel at component level*
- No dependency on model or rest of the application
- Just show/format the data that is passed through properties
- Include layout, formatting HTML and style

React and MVC

Calling setState({}) will guarantee re-rendering. Perfect for an update() method in a view/component

Implement Observer in the Container Component

The Presentation component is agnostic about the data store (MVC or redux)

The same technique can be used with Redux, if observing the store

react-redux is the more fine-tuned way of achieving the connection between React and Redux

react-redux

generates Container components

… to connect Presentation components

… with the application Redux store

Container-presentation examples: MVC and redux

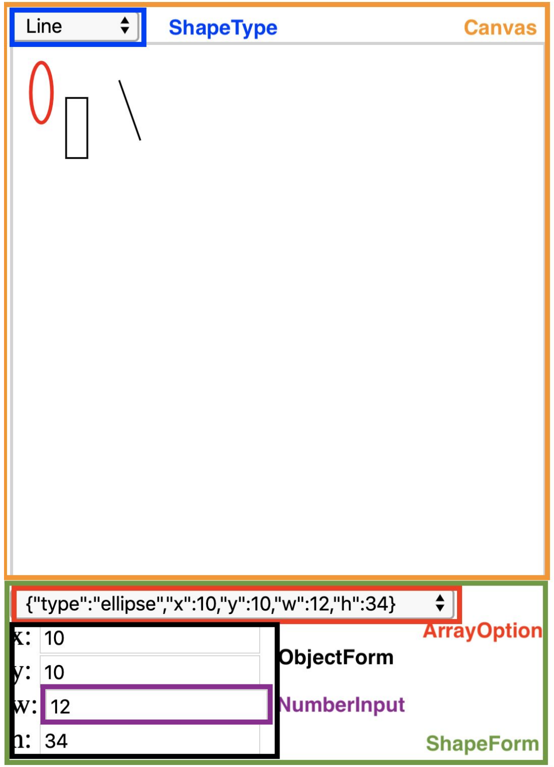
Presentation components: ShapeCanvas, ShapeForm are common and are **not** dependent on how the data/state is kept!

MVC containers of presentation components: view+controller container components:

- Shape canvas MVC container
- Shape form MVC container

“Manual” redux container of Presentation components by a Container component: Shape canvas redux container

react-redux creation of Container component: Shape form container



Line

{"type":"ellipse","x":10,"y":10,"w":12,"h":34}

x: 10
y: 10
w: 12
h: 34

React checklist

Layout **JSX** **react-dom** or **react-native** **no binding** Canvas

Populating View with ~~Model~~ hierarchical State data `<tag attr={expr} >{expr <tag2 .../> ...}</tag>`

Array data `array.map( (element, index) => <tag key={i}… {expr (element.field … index) } >)`

Conditional View population `expr && <tag… >`

Interaction <tag onEvent={callback} > Canvas

Two-way-binding? NO! Complete view re-render in virtual DOM + **reconciliation** w browser DOM

Controller separation? No (see Container vs Presentation below)

Components `class Comp extends Component { constructor(props){super(props); this.state=...}`

`render= ()=><tag…>  }`

functional: `const Comp= ({p1, p2}) => <tag ... {p1} …>` `destructuring the props object {p1, p2}`

Inputs props (immutable) this.props.children function-as-child Canvas

Outputs: new events function props Canvas

Lifecycle `getDerivedStateFromProps()` **`setState()`** **`render()`** `componentDidMount()`

Model-ViewModel connection / injection None

Container vs Presentation components (hierarchical M+Cm -- Ce+V separation) Canvas

Navigation (Router) ? Custom components, e.g. `react-router-dom`

State management ? redux store, reducers, provider, react-redux Canvas **or MVC** Canvas

Remote data support? None, just **setState()** when the data has arrived

Angular checklist

Layout
- Populating View with Model data {{expr}} <tag [property]=”expr” ..> | pipe Canvas
- Array data <tag *ngFor=”let index in collection” ..> Canvas
- Conditional View population <tag *ngIf=”boolean expr” ..> Canvas

Interaction <tag (event)="doSomething($event)" > Canvas
- Two-way-binding? <tag [(ngModel)]=”expr” > Canvas
- Controller separation? @HostListener('eventType',['$event']) doSomething(e) { ...} Canvas

Components @Component({selector:’myComp’, template:’<tag>...’}) class MyComp{..} Canvas
- Inputs @Input name; [name]=”expr” Canvas
- Outputs: new events @Output name; (name)=”expr” Canvas
- Lifecycle class MyComp implements OnInit { ngOnInit(){... } … }

Model-ViewModel connection / injection @Component({... providers:[pr1, pr2]...}) @Injectable Canvas

Navigation (Router) ? Canvas RxJS Observable get and set route Canvas

State management ? No standard, Redux can be used

Remote data support? via RxJS Observable Canvas