*Todo list

MSc in Computer Science 2017-18

Project Dissertation

Project Dissertation title: Reflectance Transformation Imaging

Term and year of submission: Trinity Term 2018

Candidate Name: Johannes Bernhard Goslar

Title of Degree the dissertation is being submitted under: MSc in Computer Science

Abstract

Vivamus vehicula leo a justo. Quisque nec augue. Morbi mauris wisi, aliquet vitae, dignissim eget, sollicitudin molestie, ligula. In dictum enim sit amet risus. Curabitur vitae velit eu diam rhoncus hendrerit. Vivamus ut elit. Praesent mattis ipsum quis turpis. Curabitur rhoncus neque eu dui. Etiam vitae magna. Nam ullamcorper. Praesent interdum bibendum magna. Quisque auctor aliquam dolor. Morbi eu lorem et est porttitor fermentum. Nunc egestas arcu at tortor varius viverra. Fusce eu nulla ut nulla interdum consectetuer. Vestibulum gravida. Morbi mattis libero sed est.

Todo Text:
Abstract

Acknowledgements

Todo Text:
Acknowledgements

Contents

List of Figures

List of Tables

1 Implementation

1.1 Overview

Since this section explains the current implementation of the developed tool set, it not only aims to fulfill the dissertation's requirements, but also to help users who want to understand the underlying systems and prepare them for potentially joining the development effort. Although abridged code extracts are of their current state for thesis submission, and the main principles will remain, nevertheless future readers are welcome to consult the actual source code if any discrepancies arise or reexport the document.

Firstly the main libraries are briefly explained where they are relevant to the program. Secondly, the mostly abstract plugin architecture is shown. Thirdly the main plugins are presented and last the delivery processes to the end users are described.

All implementation files are contained and delivered inside a single git repository, which is freely available online: `https://github.com/ksjogo/oxrti`. All following file paths are relative to that repository's root. All future development will be immediately available there and the current compiled software version is always fed automatically from it into the hosted version at `https://oxrtimaster.azurewebsites.net/api/azurestatic`.

1.2 Libraries

TypeScript

The official header line of TypeScript gives reasons why it was picked for this project: "TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open source."[27] Which fits requirements **??**, **??**. Whereas plain JavaScript would have allowed slightly easier initial on-boarding and maybe easier immediate code 'hacks', TypeScript will provide better stability in the long term and a quite improved developer experience (requirement **??**) in the long run. With the full typed hook system (compare section 1.4) it ensures that a compiled plugin will not have runtime type problems, reducing the amount of switching between code editor and the running software. The whole project is set up in a way to fully embrace editor tooling. Visual Studio Code[28] and Emacs[10] are the 'officially' tested editors of the project. Code is recommended as it will support all developer features out of the box. The installation of the tslint[25] plugin[26] is recommended to keep a consistent code style, which is configured within the *tslint.json* file. Most importantly TypeScript adds type declarations (and inference) to JavaScript, e.g.:

```
const thing = function (times: number, other: (index: number) => boolean) { ... }
```

would define *thing* as a function, taking a numbers as first argument and another function (taking a number as first parameter and returning a boolean) as second argument. The other most used TypeScript features inside the codebase are Classes[2], Decorators[5] and Generics[8], which will be discussed at their first appearance inside the code samples.

React

The two main points on React's official website are "Declarative" and "Component Based"[20], which is best shown by an extended example from their website, which exemplifies multiple patterns found through the oxrti implementation. The most important concept is the jump from having a stateful HTML document, which the JavaScript code is manipulating directly, e.g.:

```
document.getElementById('gsr').innerHTML="<p>You shouldn't do this</p>"
```

Which is diametric to requirements **??** and **??** as it would require developers to manually keep track of all data cross-references (e.g. the pan values having to automatically adapt to the current zoom level). A declarative approach instead allows much better and easier implemented reactiveness and better performance (requirements **??** and **??**) as the necessary changes can be tracked and components can be updated selectively.

```
// a class represents a single component
class Timer extends React.Component {
  // the parent component can pass on props to it
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    //  the state is updated and the component is automatically re-rendered
    this.setState(prevState => ({
      seconds: prevState.seconds + 1
    }));
  }

  // called after the component was created/added to the browser window
  componentDidMount() {
```

```
    this.interval = setInterval(() => this.tick(), 1000);
  }

  // called before the component will be deleted/removed from the browser window
  componentWillUnmount() {
    clearInterval(this.interval);
  }

  // the actual rendering code
  // html can be directly embedded into react components
  // {} blocks will be evaluated when the render method is called
  // which will happen any time the props or its internal states updates
  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

// mountNode is a reference to a DOM Node
// the component will be mounted inside that node
ReactDOM.render(<Timer />, mountNode);
```

In conjunction with mobx and TypeScript no classes are used for React components though, but instead Stateless Functional Components ('SFCs'[14]). These SFCs are plain functions, only depending on their passed properties:

```
function SomeComponent(props: any) {
 return <p>{props.first} {props.first}</p>
}
```

This component could then be used by:

```
<SomeComponent first="Hello" second="World"/>
```

This component systems allows the plugins to define some components and then 'link' them into the program via the hook system, which will be explored later.

MobX

Its main tagline is "Simple, scalable state management"[18]. An introductory overview is shown in Figure 1. Broadly speaking MobX introduces observable objects. Instead of the aforementioned DOM handling or property passing inside React trees, components can just retrieve their values from the observable objects and will be automatically refreshed if the read values change. This for example makes the implementation of the QuickPan plugin extremely easy, as it can just read the zoom, pan, etc. values of the other plugins and will automatically receive all updates without any further manual observation handling.

mobx-state-tree

"Central in MST (mobx-state-tree) is the concept of a living tree. The tree consists of mutable, but strictly protected objects"[19] This allows the implementation to have one shared state tree which can be used to safely access all data. All nodes inside the state tree are MobX observables. A simple tree with plain MST would look like this:

```
// define a model type
const Todo = types
 .model("Todo", {
  // state of every model
  title: types.string,
  done: false
 })
 .actions(self => ({
  //methods bounds to the current model instance
  toggle() {
   self.done = !self.done
  }
 }))
// create a tree root, with a property todos
const Store = types.model("Store", {
    todos: types.array(Todo)
})
```

This syntax was deemed to convoluted, as it is a lot more complex than standard JavaScript/TypeScript classes, which were introduced by the ES6 version, as shown in the React description above and thus being in conflict with requirement ??.

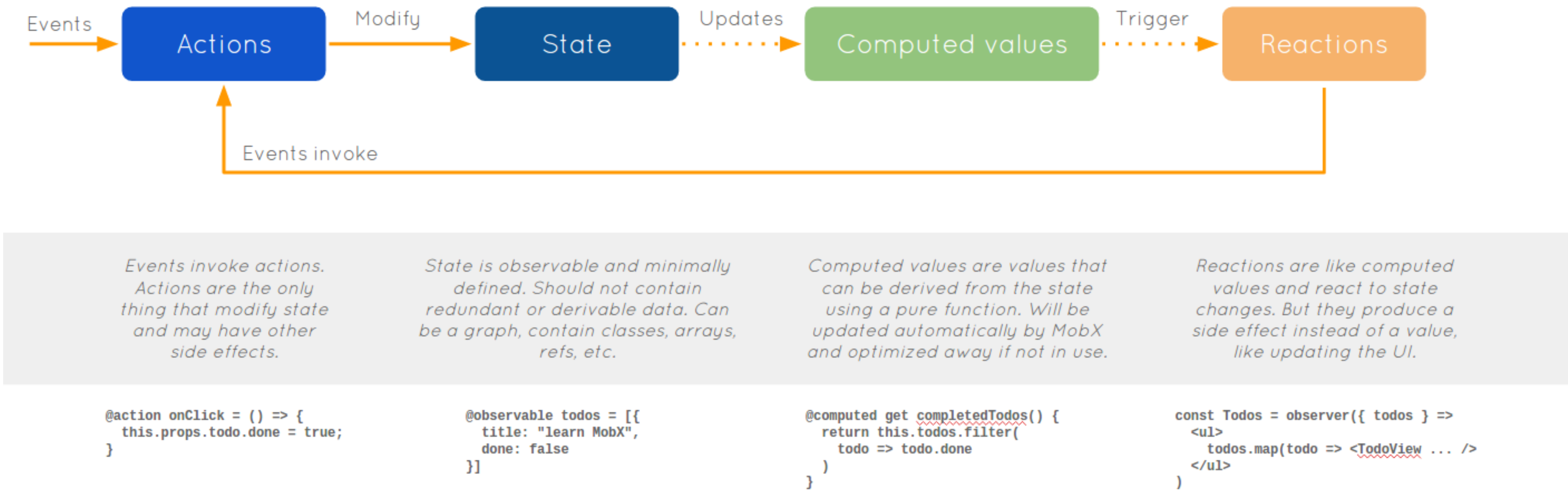

Figure 1: Taken from Weststrate[18]. Actions in the oxrti context are most often initially user actions, which are then calling into plugins to change the state. The state is mostly encapsulated on a plugin basis with usage of the mobx-state-tree library, which also encapsulates most computed values. Reactions are most often the previously discussed React components.

classy-mst

There is an option to use a more traditional syntax instead though, with the classy-mst library, with which the example above becomes[3]:

```
const TodoData = types.model({
    title: types.string,
    done: false

});

class TodoCode extends shim(TodoData) {
    @action
    toggle() {
        this.done = !this.done;
    }
}

const Todo = mst(TodoCode, TodoData, 'Todo');
```

Weststrate, the original author of MobX initially was sceptic of this syntax[1] as it was changing the semantics of ES6 classes, as classy-mst's methods will be automatically bound to the instance. This boundness is an advantage for this implementation though, as the hook configurations can just refer to `this.someMethod` instead of `this.someMethod.bind(this)`. The `@action` is a decorator, enabling the following method to change the state/properties of the model, as MST prohibits that by default. Reactions/View updates will only happen after the outermost action finished executing.

WebGL

The increasing support of the WebGL stack is the main reason why it is now feasible to implement a full RTI software stack with plain web technologies, as it "enables web content to use an API based on OpenGL ES 2.0 to perform 3D rendering in an HTML `<canvas>` in browsers that support it without the use of plug-ins."[32] OpenGL ES 2.0 likeness means that (most importantly) shaders are supported, allowing the implementation to be split up into multiple shaders with single responsibilities. For details refer to section 1.7. While preserving compatibility and requirement **??** WebGL 2 support is sadly not widespread enough to fully rely on yet (compare Figure 2), as it is currently estimated at 50% of all devices[30]. Potential improvements when WebGL 2 is more widely supported or in conditional plugins are discussed in **??**. One notable limitation of WebGL is `MAX_TEXTURE_IMAGE_UNITS`, the maximum amount of bound textures inside a single shader, which in most implementations is 16[31], whereas the standard OpenGL implementations are likely to have a limit of

Desktop Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	9	(Yes)	4.0 (2.0)	11	12	5.1
WebGL 2	56	No support	51 (51)	No support	43	No support

Desktop **Mobile**

Feature	Chrome for Android	Edge	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	25	(Yes)	4	No support	12	8.1
WebGL 2	?	?	?	?	?	?

Figure 2: WebGL compability as from the Mozilla Developer network[24].

32. This is influencing the BTF file format, as for example in the PTM RGB use case a total of 18 coefficients exist, which now need to be bundled up somehow into maximum 16 textures, if the calculations should be done inside a single shader. It is also limiting the amount of layers of the Paint plugin, as these also consist of bound textures. Apart from the shaders, which are written in the OpenGL ES Shading Language[29] and the the texture loader (section 1.8), no direct WebGL code is necessary nor used anywhere inside the implementation, as the gl-react library is abstracting it neatly for use from the MobX/React environment.

gl-react

"Implement complex effects by composing React components."[22] is the main use of the gl-react library. A minimal component, adapted from the gl-react-cookbook looks like[22]:

```
const shaders = Shaders.create({
  helloGL: {
    frag: GLSL`
     precision highp float;
     varying vec2 uv;
     void main() {
      gl_FragColor = vec4(uv.x, uv.y, 0.5, 1.0);
    }`
  }
});

export default class Example extends Component {
  render() {
    return (
      <Surface width={300} height={300}>
        <Node shader={shaders.helloGL} />
      </Surface>
    );
```

Figure 3: RGBA texture, with R and G according to their respective u or v texture coordinate. From [9].

```
  }
}
```

Which would result in a display like Figure 3. gl-react's is not a 3D engine, so no objects are to be created or scene graph managed, instead the oxrti implementation can concentrate on solely providing the necessary shaders. gl-react's default node size is taken from the parent surface size. The surface size will be dependent on the user running the program and his browser windows, which makes it undesirable as details would be lost, if the BTF provided more detail, so the processing Node sizes are usually set to the BTF resolution or higher.

Webpack

Webpack is used to bundle the implementation into single files as it is "a bundler for javascript and friends. Packs many modules into a few bundled assets."[33] A more detailed discussion on the targets is in section 1.10. Broadly speaking Webpack loads the source code inside the *src* directory according to the loaders defined inside the *webpack.config.js* file, analyses their dependencies and then bundles them together. This makes it possible to have a dependency tree spanning 26184 packages from npm, but still providing a single bundled application file only 1.5 megabyte large (data as of August 26, 2018). It also allows the dynamic plugin structure by bundling the plugins into a dynamic 'context' from which single plugins can be loaded at runtime.

Electron

Electron is used to "build cross-platform desktop apps with JavaScript, HTML, and CSS"[6] While theoretically not necessary to fulfill most requirements, as the implementation is compatible with all modern web browsers, an additional standalone executable provides some advantages:

- It is possible to add a more traditional menu-based interface, which the browser version could not support.
- Stable development environment, as electron-devtools-installer is used to provide relevant extensions (React devtools, MobX devtools) by default and the hot reloading is reliably tested, which together form a good developer experience (requirement **??**)
- It allows to preserve the software in a usable, contained state, not relying on the user also having a compatible web browser in the future.
- It allows future development to more directly access resources of the host machine, e.g. the normal OpenGL stack could be used for calculating the coefficients, as it is less resource constrained compared to the WebGL stack.

MaterialUI

MaterialUI is succinctly described by "React components that implement Google's Material Design."[17]. MaterialUI's component are used throughout the app for styling the components, making the use of custom CSS largely unnecessary apart from minor positioning fixes. For example the Zoom component is defined as:

```
// Card, CardContent, Tooltip and Button are all components provided by MaterialUI.
// this refers to the Zoom Plugin's controller which
// the content will be automatically refreshed if the refered values change
const Zoom = Component(function ZoomSlider (props) {
    return <Card style={{ width: '100%' }} >
        <CardContent>
            <Tooltip title='Reset'>
                <Button onClick={this.resetZoom} style={{ marginLeft: '-8px' }}>Zoom</Button>
            </Tooltip>
            <Tooltip title={this.scale}>
                <Slider value={this.scale} onChange={this.onSlider} min={0.01} max={30} />
            </Tooltip>
            <Tooltip title='Reset'>
```

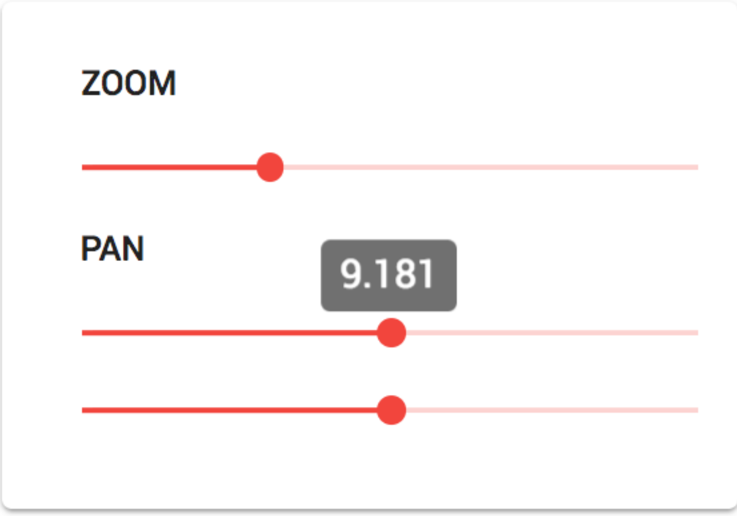

Figure 4: Zoom Component, the user is dragging the zoom slider currently, the mouse pointer is not depicted.

```
                <Button onClick={this.resetPan} style={{ marginLeft: '-11px' }}>Pan</Button>
            </Tooltip>
            <Tooltip title={this.panX}>
                <Slider value={this.panX} onChange={this.onSliderX} min={-1 * this.scale} max={1 * this.scale} />
            </Tooltip>
            <Tooltip title={this.panY}>
                <Slider value={this.panY} onChange={this.onSliderY} min={-1 * this.scale} max={1 * this.scale} />
            </Tooltip>
        </CardContent>
    </Card>
})
```

It would result in a display like Figure 4.

Misc

Further libraries of note are:

electron-webpack[7] is providing the bridging between Webpack and Electron, its config is expanded by the *webpack.renderer.additions.js* and *webpack.renderer.shared.js* files.

pngjs[23] is providing in-browser bitwise png manipulation, required in the converter.

jszip[15] is providing in-browser zip file manipulations, which are fundamental for the BTF fileformat.

1.3 Base

The implementation is making a distinction between plugin and non-plugin files. The amount of non-plugin files was aimed for to be as low as possible, as they are inflexible in all output configurations and will have slightly different behaviour while developing in regard to reloads. The files not contained in plugins are the following:

- *AppState.tsx*, the mobx-state-tree root node representing the whole application state in its leafs, detailed in section 1.6.
- *BTFFile.tsx*, containing the fileformat implementation and utility functions, described in section 1.5.
- *Hook.tsx* and *HookManager.tsx* which provide the whole dynamic interaction system between the different plugins, shown in section 1.4.
- *Loader.tsx*, *electron/index.tsx*, *renderer/index.tsx* and *web/index.tsx*, providing the loading functionality. The Electron application has two entry points, one for the main process, which is *electron/index.tsx* and one for the in-browser content, which is *renderer/index.tsx*. The in-browser one and the plain browser entry point *web/index.tsx* both call the *Loader.tsx* to initialise the state management and mount the root React component, so the user can interact finally. The Loader also handles hot-reloading, it will receive the changed source code from Webpack and update the plugins accordingly.
- *Plugin.tsx* defining the base class for a plugins, further explained in section 1.9.
- *types.d.ts* is providing the custom ambient type declarations for software dependencies, which are not providing TypeScript types on their own. In case new dependencies are added, they are likely to require and addition there.
- *Util.tsx* providing general helper functions, largely related to some math functions for texture coordinate handling.
- *loaders/glslify-loader/index.js* is the custom Webpack loader for *.glsl* files, allowing e.g. the `import zoomShader from './zoom.glsl'` statement and setting up Webpack to contain the shader source in the final bundle.

- *loaders/oxrtidatatex/OxrtiDataTextureLoader.tsx* is providing direct texture loading from in-memory BTF files, discussed in section 1.8.

1.4 Hooks

The hook system allows stable and prioritized interactions between the different plugins. All available hooks are declared inside the Hook.tsx file, which offers 3 different types of hooks:

```
// Hooks are sorted in descending priority order in their respective `HookManager`
export type HookBase = { priority?: number }

// Generic single component hook, usually used for rendering a dynamic list of components
export type ComponentHook<P = PluginComponentType> = HookBase & { component: P }

// Generic single component hook, usually used for notifications
export type FunctionHook<P = (...args: any[]) => any> = HookBase & { func: P }

// Generic hook config, requiring more work at the consumer side
export type ConfigHook<P = any> = HookBase & P

// union of all hooks to allow for manual hook distinction
export type UnknownHook = ComponentHook & FunctionHook & ConfigHook

// object of named hooks
type Hooks<P> = { [key: string]: P }

// collection of unknown hooks
export type UnknownHooks = Hooks<UnknownHook>

// hook configuration inside plugins: 1-Hookname->*-LocalName->1-HookConfig
export type HookConfig = { [P in keyof HookTypes]: Hooks<HookTypes[P]> }

// all hooknames
export type HookName = keyof HookConfig
```

```

// map one hookname to its type
export type HookType<P extends HookName> = HookTypes[P]

// list of hooknames inside hook collection T, having hooktype U
type LimitedHooks<T, U> = ({ [P in keyof T]: T[P] extends U ? P : never })[keyof T]

// limit hookname parameters to a type conforming subset, e.g. LimitedHook<ComponentHook>
export type LimitedHook<P> = LimitedHooks<HookConfig, Hooks<P>>
```

These types are used to first declare single hook types (which will be discussed within the plugins consuming them) and then construct the whole hook configuration tree for all plugins:

```
type HookTypes = {
  ActionBar?: ConfigHook<ActionBar>,
  AfterPluginLoads?: FunctionHook,
  AppView?: ComponentHook,
...
}
```

A plugin then can link itself into these hooks with its hooks method, for example:

```
get hooks () {
  return {
    // register things for the ViewerSide hook / add components to the side bar
    ViewerSide: {
      // a plugin can register itself multiple times with different names and configurations
      Metadata: {
        component: BTFMetadataConciseDisplay,
        // hooks will be sorted internally in priority order, highest first
        priority: -110,
      },
      Open: {
        component: Upload,
        priority: 100,
      },
    },
  }
```

```
}
```

The state manager (section 1.6) collects all hooks and merges them into the respective HookManagers, which are then used to iterate/map over these:

```
/** type definitions for the different iterators */
export declare type HookIterator<P extends HookName> = (hook: HookType<P>, fullName: string) => boolean | void;
export declare type AsyncHookIterator<P extends HookName> = (hook: HookType<P>, fullName: string) => Promise<boolean | void>;
export declare type HookMapper<P extends HookName, S> = (hook: HookType<P>, fullName: string) => S;
export declare type HookFind<P extends HookName, S> = (hook: HookType<P>, fullName: string) => S;
/** * Manage one named hook */
export declare class HookManagerCode extends ShimHookManager {
    /**
     * Add some hook into the managed stack
     * @param name in `Plugin£Hookname£Entryname` form
     * @param priority higher will be treated first with the iterators
     */
    insert(name: string, priority?: number): void;
    /** Iterate with iterator over all registered hooks, stop iteration if the iterator is returning true, name is redundant as it could be inferred from ourselves, but allows for easy typesafe calling, appState is needed to retrieve the current plugin instance */
    forEach<P extends HookName>(iterator: HookIterator<P>, name: P, appState: IAppState): void;
    /** iterate over all hooks, but wait for asynchronous hooks to finish before executing the next one */
    asyncForEach<P extends HookName>(iterator: AsyncHookIterator<P>, name: P, appState: IAppState): Promise<void>;
    /** iterate in reverse order */
    forEachReverse<P extends HookName>(iterator: HookIterator<P>, name: P, appState: IAppState): void;
    /** map over all hooks */
    map<S, P extends HookName>(mapper: HookMapper<P, S>, name: HookName, appState: IAppState): S[];
    /** get the concrete hook at index number */
    pick<P extends HookName>(index: number, name: P, appState: IAppState): HookType<P>;
```

```
}
```

1.5 BTF File

The full standalone BTF file format specification can be found inside Appendix A. The implementation in *BTFFile.tsx* is an in-memory implementation of that file with following interface, it is mainly a 'dumb' data container.

```
export default class BTFFile {
    /** running id numbers to allow easy cache busts */
    id: number;
    /** JSON object of the included oxrti state */
    oxrtiState: object;
    /** default data representation */
    data: Data;
    /** reference to annotation layers */
    layers: AnnotationLayer[];
    /** user visible name */
    name: string;
    /** manifest can come from an unpacked zip, usully typed as any */
    constructor(manifest?: BTFFile);
    /** cannocical zip name for name and id */
    zipName(): string;
    /** return true if no data is contained/is dummy object */
    isDefault(): boolean;
    /** export the JSON data of the manifest.json file */
    generateManifest(): string;
    /** export user visible shortened metadata */
    conciseManifest(): string;
    /**
     * Generate a unique tex container which the gl-react loader will cache
     * @param channel reference to the named channel
     * @param coefficent reference to the named child coefficent of channel
     */
    texForRender(channel: string, coefficent: string): TexForRender;
    /**
     * Generate a tex configuration for a layer
     * @param id of the layer, must be found in this.layers
     */
```

```
    annotationTexForRender(id: string): TexForRender;
    /** aspect ratio of the contained data */
    aspectRatio(): number;
    /** package the current data into a zip blob */
    generateZip(): Promise<Blob>;
}
/** unpackage a zip blob into a BTFFile */
export declare function fromZip(zipData: Blob | ArrayBuffer):
 ↪  Promise<BTFFile>;
```

Notable is the `PreDownload` hook which is called before the `generateZip` function is called, to allow the ImpExp plugin and the Paint plugin to fill the BTF with their respective data. This hook is called by the AppState though, as the BTFFile implementation is fully standalone (apart from the jszip dependency), in case other software wishes to re-use the implementation.

1.6 State Management

The *AppState.tsx* file is the root of the applications state tree, but itself only contains limited data:

```
// types refering to MST types
const AppStateData = types.model({
  // keep references to loaded plugins
  plugins: types.late(() => types.optional(types.map(Plugin),
   ↪  {})),
  // have a HookMangager for each HookName
  hooks: types.optional(types.map(HookManager), {}),
  // currently opened BTF file, name is the key for the
   ↪  BTFCache
  currentFile: '',
})
```

Its main function is to load all plugins and then let them share data between each other via the hook system. The set of plugins to be loaded is defined inside *oxrti.plugins.json*, the plugins defined there will be loaded in order. The state tree after inital plugin load is shown in Figure 6. As only `@actions` can modify the state, it is easy to follow the changing app state, as shown in Figure 5.

1.7 Renderer Stack

Even though the renderer stack is fully contained in the plugins, its principles span most of them, thus an overview is in order. All WebGL rendering is done

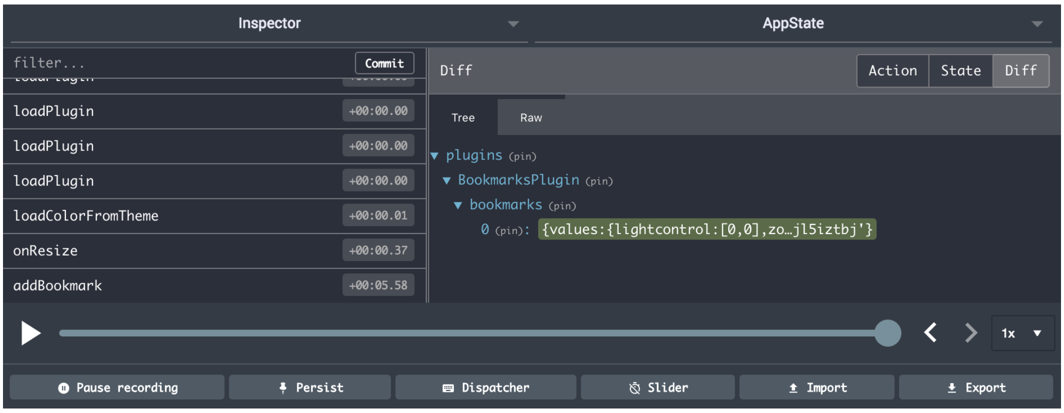

Figure 5: Run MobX at application startup and after one click on 'Add Bookmark'. The differential of that action is on the right.

through the gl-react library. If a plugin wants to register a node inside the stack, it is using the hook system, e.g. the Rotation plugin:

```
// register two nodes inside the rendering stack
// higher priority will be run first
ViewerRender: {
    // first center the underlying texture
    Centerer: {
        component: CentererComponent,
        inversePoint: this.undoCurrentCenterer,
        forwardPoint: this.doCurrentCenterer,
        priority: 11,
    },
    // then rotate it
    Rotation: {
        component: RotationComponent,
        inversePoint: this.undoCurrentRotation,
        forwardPoint: this.doCurrentRotation,
        priority: 10,
    },
},
```

The registered nodes are components again, which will link them into the automated mobx reactions, for example the Centerer node:

```
export const CentererComponent = Component(function CentererNode
    (props) {
    // dynamic sizes depending on the loaded btf
    // if the btf changes, the uniforms will be updated
        automatically
    let [width, height] = this.centererSizes
```

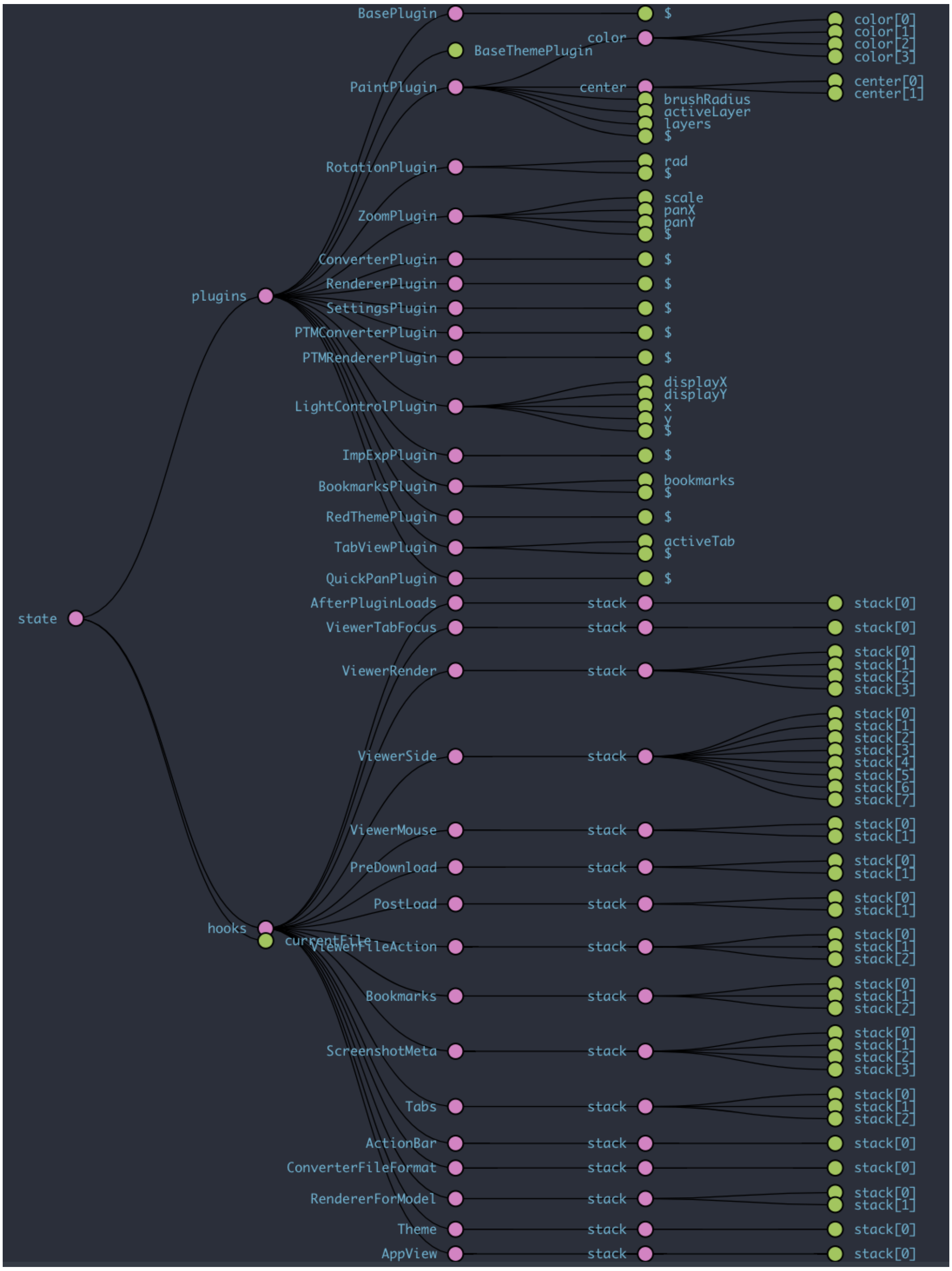

Figure 6: State tree after inital application load. All plugins are initialized and their hooks are registered. Visualized with redux dev tools.[21]

```
    let maxDims = this.maxDims
    // create one gl-react Node
    return <Node
        width={maxDims}
        height={maxDims}
        shader={{
            // shader from import centerShader from './centerer.glsl'
            frag: centerShader,
        }}
        // uniforms will be automatically type-converted to the appropiate WebGL types
        uniforms={{
            // refering to rendering output one step before
            children: props.children,
            inputHeight: height,
            inputWidth: width,
            maxDim: maxDims,
        }} />
})
```

For the currently delivered default configuration the following nodes are registered:

1. PaintNode
2. CentererNode
3. RotationNode
4. ZoomNode

The Renderer plugin will start by picking the proper base rendering node depending on the channel format of the currently loaded BTF file (btf inside the code):

```
props.appState.hookForEach('RendererForModel', (Hook) => {
    if (Hook.channelModel === btf.data.channelModel) {
        current = <Hook.node
            key={btf.id}
            lightPos={lightControl.lightPos}
            renderingMode={this.renderMode}
        />
    }
})
```

This inital node will then be wrapped by the registered nodes:

```
props.appState.hookForEach('ViewerRender', (Hook) => {
```

```
    current = <Hook.component
        key={btf.id}
    >{current}</Hook.component>
})
```

1.8 Texture Loader

The bridge between the loaded BTF files and the WebGL contexts needs to be closed with custom code, as gl-react was not supporting the load of PNG textures from memory. An abridged version of the code follows, as it is a good example of the Promise pattern used in some following parts.

```
loadTexture(config: TexForRender) {
    // keep track of the amount of currently loading textures
    appState.textureIsLoading()
    // shortcut for the WebGL reference
    let gl = this.gl
    // access the raw data buffer from the texture configuration
    let data = config.data
    // create a Promise, that basically is chaining callbacks together and then asynchronously executing each step
    let promise =
        // first create an ImageBitmap object, we need to flipY as the textures
        are orientated naturally inside the BTF file but WebGL is expecting them bottom row first
        createImageBitmap(data, { imageOrientation: 'flipY' })
        // the catch step is called if the previous part failed
        .catch((reason) => {
            // some browsers do not like loading a lot of big textures in parallel and will garbage collect them in between and thus fail, as a fallback the limiter function is used to limit concurrency of that part
            return limiter(() => createImageBitmap(data))
        })
        // the then part is called when the previous step succeeded
        .then(img => {
            // create and bind a new WebGL texture
            let texture = gl.createTexture()
            gl.bindTexture(gl.TEXTURE_2D, texture)
            let type: number
```

```
                // map the type from the BTF file to a WebGL texture type
                switch (config.format) {
                    case 'PNG8':
                        type = gl.LUMINANCE
                        break
x                    // ...
                    case 'PNG32':
                        type = gl.RGBA
                        break
                }
                // load the imageBitmap into the texture
                gl.texImage2D(gl.TEXTURE_2D, 0, type, type, gl.UNSIGNED_BYTE, img)
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST)
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST)
                // finished and return
                appState.textureLoaded()
                return { texture, width: img.width, height: img.height }
            })
            .catch((reason) => {
                // a null texture will be empty
                alert('Texture failed to load' + reason)
                appState.textureLoaded()
                return { texture: null, width: config.width, height: config.height }
            })
    // the calling code will have its own catch/then logic
    return promise
}
```

1.9 Plugins

All plugins extend the relatively limited abstract plugin class in *Plugin.tsx* and have no preserved state:

```
/** General Plugin controller, will be loaded into the MobX state tree */
export declare class PluginController extends PluginShim {
    /** referential access to app state, will be set by the plugin loader */
```

```
    appState: IAppState;
    /** called when the plugin is initally loaded from file */
    load(appState: IAppState): void;
    /** all hooks the plugin is using */
    readonly hooks: HookConfig;
    /** get a single typed hook */
    hook<P extends HookName>(name: P, instance: string): HookType<P>;
    /** called before the plugin will be deleted from the state tree, ususally used for volatile state fixes, e.g. paint layers */
    hotUnload(): void;
    /** called after the plugin was restored in the state tree */
    hotReload(): void;
    /** convenience function to inverse a rendering point from surface coordinates into texture coordinates */
    inversePoint(point: Point): Point;
    /** some components need references to their actual DOM nodes, these are stored outside the plugins scope to allow hot-reloads */
    handleRef(id: string): (ref: any) => void;
    /** return a stored ref */
    ref(id: string): any;
}
```

The most important function is the `PluginCreator`, which merges the mobx-state-tree model with the classy-mst controller code and provides a wrapper function to create components bound to the containing plugin:

```
/**
 * Create Subplugins
 * @param Code is the controller
 * @param Data is the model
 * @param name must be the same as the folder and filename
 */
function PluginCreator<S extends ModelProperties, T, U> (Code: new () => U, Data: IModelType<S, T>, name: string) {
    // create the resulting plugin class
    let SubPlugin = mst(Code, Data, name)
    // higher-order-component
    // inner is basically (props, classes?) => ReactElement
    // inner this will be bound to the SubPlugin instance
```

```
    type innerType<P, C extends string> = (this: typeof SubPlugin.Type, props: ComponentProps & { children?: ReactNode } & P, classes?: ClassNameMap<C>) => ReactElement<any>
    // P are they freely definable properties of the embedded react component
    // C are the inffered class keys for styling, usually no need to manually pass them
    function SubComponent<P = {}, C extends string = ''> (inner: innerType<P, C>, styles?: StyleRulesCallback<C>): PluginComponentType<P> {
        // wrapper function to extract the corresponding plugin from props into plugin argument typedly
        let innerMost = function (props: any) {
            let plugin = (props.appState.plugins.get(name)) as typeof SubPlugin.Type
            // actual rendering function
            // allow this so all code inside a plugin can just refer to this
            let innerProps = [props]
            // append styles
            if (styles)
                innerProps.push(props.classes)
            // call the embedded component
            return inner.apply(plugin, innerProps)
        };
        // set a nice name for the MobX/redux dev tools
        (innerMost as any).displayName = inner.name
        // use MobX higher order functions to link into the state tree
        let func: any = inject('appState')(observer(innerMost))
        // wrap with material-ui styles if provided
        if (styles)
            func = withStyles(styles)(func);
        // also name the wrapped function for dev tools
        (func as PluginComponentType<P>).displayName = `PluginComponent(${inner.name})`
        return func
    }
    // allow easier renaming in the calling module
    return { Plugin: SubPlugin, Component: SubComponent }
}
```

A minimal example would be:

```
const BasePluginModel = Plugin.props({
    greeting: 'In the beginning was the deed!',
})

class BasePluginController extends shim(BasePluginModel, Plugin) {
    @action
    onGreeting (event: any) {
        this.greeting += '!'
    }
}

// general plugin template code
const { Plugin: BasePlugin, Component } = PluginCreator(BasePluginController, BasePluginModel, 'BasePlugin')
export default BasePlugin
// export the type to allow other plugins to retrieve this plugin
export type IBasePlugin = typeof BasePlugin.Type

// CSS styles, classnames will be mangled, so styles is passed to the component
const styles = (theme: Theme) => createStyles({
    hello: {
        color: 'red',
    },
})

// props are standard react props, classes contains the mangled names
const HelloWorld = Component(function HelloWorld (props, classes) {
    return <p className={classes.hello} onClick={this.onGreeting}>{this.greeting}</p>
}, styles)
```

The rest of the plugins are presented on a higher conceptual level, as their internal APIs are most times used only by themselves. Their hooks will be discussed though.

1.9.1 Base Plugin

The base plugin is containing no further internal logic, but only is providing some shared display components. These could theoretically have been implemented outside of any plugin, but putting them into the Base plugin simplifies the distinction between components bound to plugins and unbound plugins, by having no unbound plugins at all. It also simplifies the code loading, as just the Base plugin can be (re-)loaded like all other plugins. The provided components are:

- JSONDisplay, showing a JSON object in a prettified form, used for displaying different metadata objects
- BTFMetadataDisplay, showing the currently loaded BTF's data
- RenderHooks, to render all components attached to a hook
- SafeGLIInspector, wrapping the secondary WebGL surfaces and disabling theme, if WebGL debug tools are used, which can handle only one surface.
- Tooltip, wrapping the material-ui tooltip component to fix styling errors

1.9.2 BaseTheme, RedTheme and BlueTheme Plugin

The BaseTheme plugin is having two essential properties:

```
/** app wide theme definitions */
themeBase: ThemeOptions = {
    palette: {
    },
    overrides: {
        MuiTooltip: {
            tooltip: {
                fontSize: 16,
            },
            tooltipPlacementBottom: {
                marginTop: 5,
            },
            tooltipPlacementTop: {
                marginBottom: 5,
            },
        },
    },
}

```

```
/** per theme plugin overridable definitions */
themeExtension: ThemeOptions = {}
```

Concrete themes (at the moment RedTheme and BlueTheme plugins) extend this BaseTheme and then set their `themeExtension` according to their modifications. To pick a theme to be used the ThemeConfig hook exists:

```
type ThemeConfig = {
    controller: { theme: Theme },
}
```

Concrete themes register with:

```
get hooks (): HookConfig {
    return {
        Theme: {
            Red: {
                priority: 100,
                controller: this,
            },
        },
    }
}
```

If multiple plugins are registering themes, the appState will pick the theme with the highest priority to apply to the app.

1.9.3 TabView Plugin

The TabView plugin is providing the full app experience and is targeted at the Electron output and the online hosted version. As a tabbed container will occasionally delete the content of non-active tabs, some hooks are needed to ensure graceful behaviour of all displayed tabs:

```
// register a new tab
type Tab = {
    // component to be the base of the tab
    content: PluginComponentType
    tab: TabProps,
    padding?: number,
    // async functions to allow customisation before/after tabs
    ↪ change
    beforeFocusGain?: () => Promise<void>,
    afterFocusGain?: () => Promise<void>,
    beforeFocusLose?: () => Promise<void>,
    afterFocusLose?: () => Promise<void>,
}
```

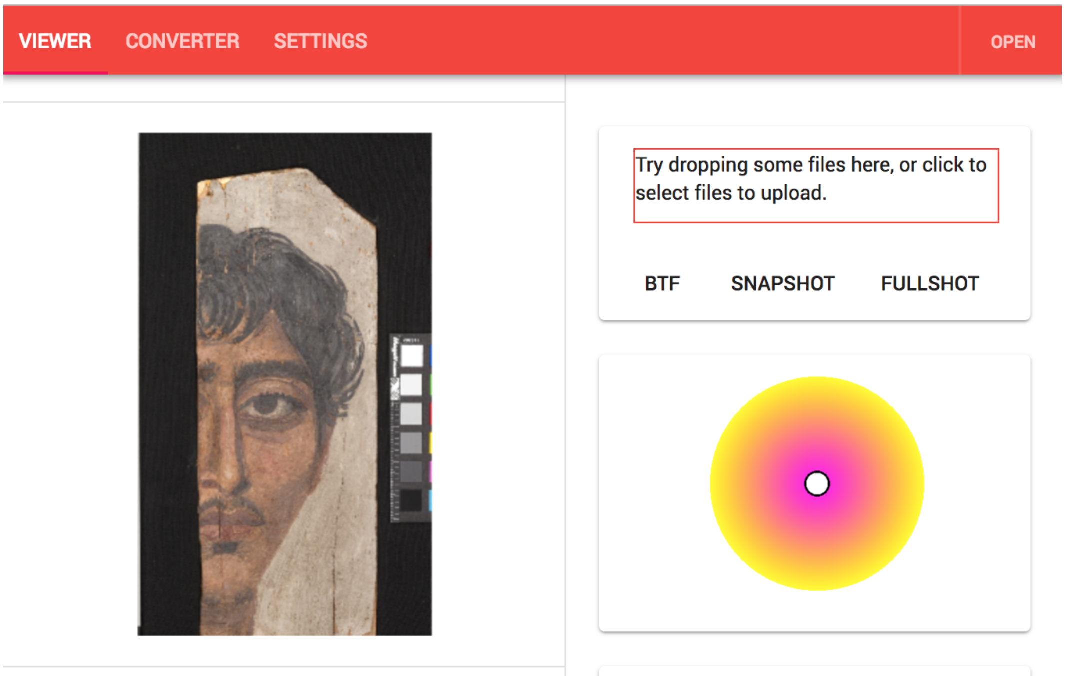

Figure 7: Viewer with loaded TabView and RedTheme plugins. Tabbar on top, with action buttons on top right.

```

// action buttons on the top rights
type ActionBar = {
    onClick: () => void,
    title: string,
    enabled: () => boolean,
    tooltip?: string,
}

// notifications if the tab changes for sub-components which are not being a tab themselves
type ViewerTabFocus = {
    beforeGain?: () => void,
    beforeLose?: () => void,
}
```

This view is depicted in Figure 7.

1.9.4 SingleView Plugin

The SingleView plugin is aimed to provide a contained viewer experience, e.g. on a museum's website, it is just displaying the tab with the highest priority, see Figure 8.

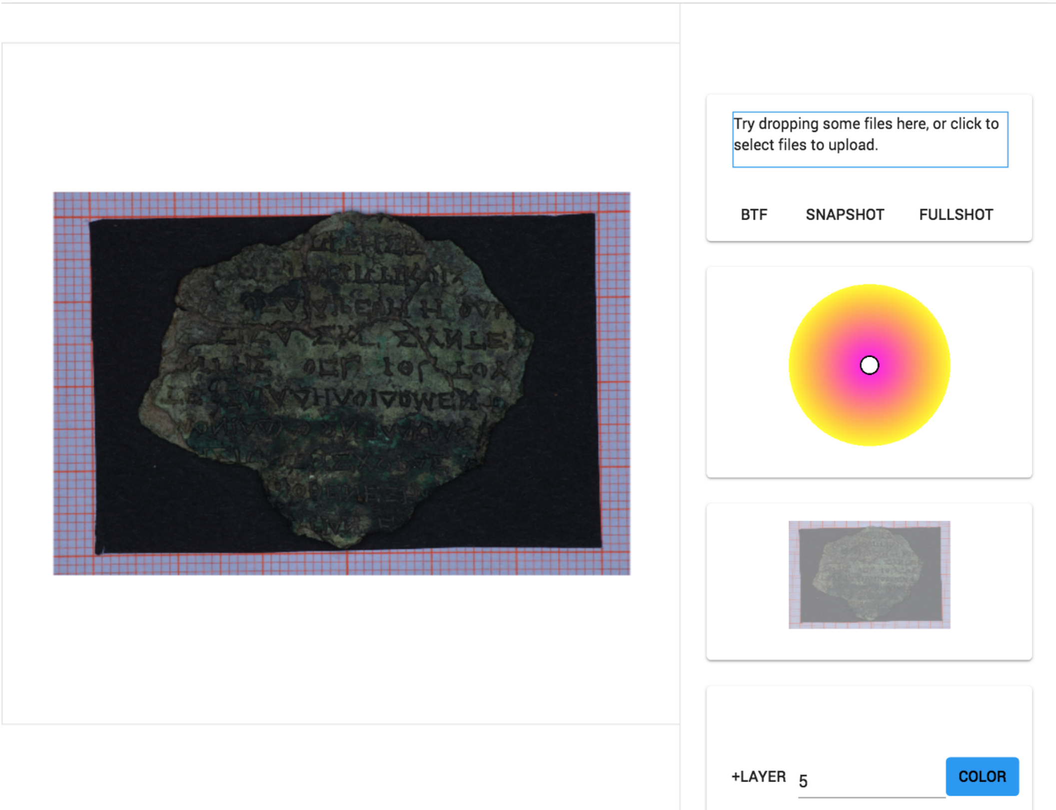

Figure 8: Viewer with loaded SingleView and BlueTheme plugins. Only changes to Figure 7 are two lines in *oxrti.plugins.json*.

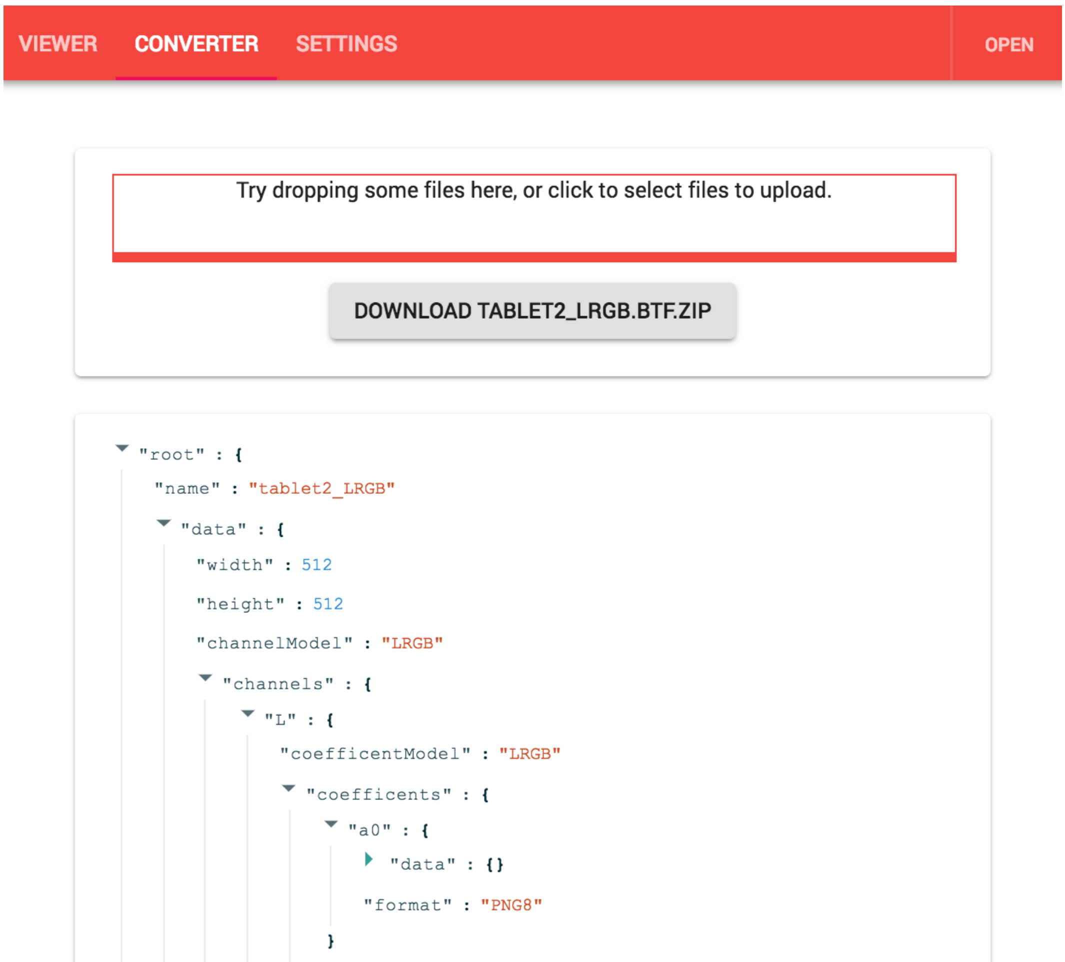

Figure 9: Converter user interface, with display of the extract manifest.

1.9.5 Converter Plugin

The Converter plugin consists of multiple parts:

- The converter user interface, as shown in Figure 9.
- BMPWriter and PNGWriter, both extending Writer to write the converted textures. The BMPWriter is customised as no current package is offering the required functionality. The PNGWriter is wrapping pngjs.
- An abstract base converter strategy, to be extended by plugins providing a concrete converter with the following interface:

```
export default abstract class ConverterStrategy {
    /** raw file buffer */
    fileBuffer: ArrayBuffer;
    /** wrapped file buffer */
    inputBuffer: Buffer;
```

```
    /** UI delegate for status updates */
    ui: IConverterUI;
    /** pixelData buffer, pointing into fileBuffer+metadata length */
    pixelData: Buffer;
    /** extracted width and height */
    width: number;
    height: number;
    /** concrete BTF output */
    output: BTFFile;
    /** freeform format depending JSON object, e.g. biases for PTMs */
    formatMetadata: object;
    /** current channelModel as defined in the BTF specification */
    channelModel: ChannelModel;
    /** total pixel count */
    readonly pixels: number;
    /** started from the converter ui */
    constructor(content: ArrayBuffer, ui: IConverterUI);
    /** pointer into the raw file buffer */
    currentIndex: number;
    /** read metadata till newline */
    readTillNewLine(): string;
    /** read one item, usually byte */
    readOne(): number;
    /** prepare pixeldata buffer */
    preparePixelData(): Promise<void>;
    /** run the actual conversion */
    process(): Promise<BTFFile>;
    /** only these need to implemented by concrete converters */
    /** read the metadata block */
    abstract parseMetadata(): Promise<void>;
    /** read the pixel block */
    abstract readPixels(): Promise<void>;
    /** read potential suffix after pixel block */
    abstract readSuffix(): Promise<void>;
    /** bundle the read pixels into channels according to the channelModel */
    abstract bundleChannels(): Promise<Channels>;
}
```

If a plugin wants to register a concrete converter it would use following hook:

```
ConverterFileFormat: {
    PTM: {
        fileEndings: ['.ptm'],
        // refering to the class extending the base strategy
        strategy: PTMConverterStrategy,
    },
},
```

1.9.6 PTMConverter Plugin

The PTM Converter plugin is converting *.ptm* files, as they are described in **??**. Currently the RGB and LRGB lightening models are supported. They main interesting part is the pixel data reader:

```
async readPixelsLRGB () {
    // allocate output buffers for ['a_0', 'a_1', 'a_2', 'a_3', 'a_4', 'a_5', 'R', 'G', 'B']
    this.coeffData = this.coeffNames.map(e => Buffer.alloc(this.pixels))
    // this.pixelData contains pixels * [a_0, a_1, a_2, a_3, a_4, a_5] and then pixels * [R, G, B]
    for (let y = 0; y < this.height; ++y)
        for (let x = 0; x < this.width; ++x) {
            // iterate over pixels
            let originalIndex = ((y * this.width) + x)
            // flip the Y index, as PTM is bottom-row first, whereas BTF is top-row first
            let targetIndex = ((this.height - 1 - y) * this.width) + x
            // read from the coefficent block
            for (let i = 0; i <= 5; i++)
                this.coeffData[i][targetIndex] = this.pixelData[originalIndex * 6 + i]
            // read from RGB block by skipping over the coefficent block and then iterating colors
            for (let i = 0; i <= 2; i++)
                this.coeffData[i + 6][targetIndex] = this.pixelData[this.pixels * 6 + originalIndex * 3 + i]
            // update the progress bar
            if (originalIndex % (this.pixels / 100) === 0) {
                await this.ui.setProgress(originalIndex / this.pixels * 100 + 1)
            }
```

```
        }
}

async readPixelsRGB () {
    // allocate output buffers for ['R0R1R2', 'R3R4R5', 'G0G1G2', 'G3G4G5', 'B0B1B2', 'B3B4B5']
    this.coeffData = this.coeffNames.map(e => Buffer.alloc(this.pixels * 3))
    // this.pixelData contains a block of  pixels * [a_0, a_1, a_2, a_3, a_4, a_5] for each color
    for (let y = 0; y < this.height; ++y) {
        for (let x = 0; x < this.width; ++x) {
            for (let color = 0; color <= 2; color++) {
                // iterate over pixels and colors
                let inputIndex = (((y * this.width) + x) + this.pixels * color) * 6
                // flip the Y index, as PTM is bottom-row first, whereas BTF is top-row first
                let targetIndex = (((this.height - 1 - y) * this.width) + x) * 3
                // iterate over the coefficents for the given pixel/inputIndex */
                for (let i = 0; i <= 5; i++) {
                    let bucket = color * 2 + Math.floor(i / 3)
                    this.coeffData[bucket][targetIndex + (i % 3)] = this.pixelData[inputIndex + i]
                }
            }
        }
        // update the progress bar
        if (y % (this.height / 100) === 0) {
            await this.ui.setProgress(y / this.height * 100 + 1)
        }
    }
}
```

1.9.7 Renderer Plugin

The Renderer plugin is providing the main user interface, which is split into two parts (compare Figure 7), on the left the rendered object and on the right further controls. As such it is using a comprehensive set of hooks:

```
// specific channelModel renderers can register their base node
type BaseNodeConfig = {
    channelModel: ChannelModel,
    // a basenode can declare that it supports multiple rendering modes e.g. default, surface normals, specular enhancement, etc.
    renderingModes: string[],
    node: PluginComponentType<BaseNodeProps>,
}

// hook for a node in the renderer stack
type RendererNode = {
    component: PluginComponentType,
    // if the node is transforming the texture coordinates, and inverse method needs to be provided
    inversePoint?: (point: Point) => Point,
    forwardPoint?: (point: Point) => Point,
}

// hook for listening to mouse event inside the main renderer
type MouseConfig = {
    dragger: (oldTex: Point, nextTex: Point, oldScreen: Point, nextScreen: Point, dragging: boolean) => boolean,
    mouseUp?: (nextScreen: Point, nextTex: Point) => boolean,
    mouseLeft?: () => void,
}

// hook for adding file actions/buttons below the upload field
type ViewerFileAction = {
    tooltip: string,
    text: string,
    action: () => Promise<void>,
}

// hook for adding information to the metadata file when shots are exported
type ScreenshotMeta = {
    key: string,
    fullshot?: () => (string | number)[] | string | number,
    snapshot?: () => (string | number)[] | string | number,
}

// components to be rendered inside the drawer
```

```
type ViewerSide = ComponentHook

// notifications, that a btf file will be exported and plugins should update their respective data inside the current in-memory version
type PreDownload = FunctionHook

// notification that a btf file was loaded, plugins can import extra data
type PostLoad = FunctionHook

// components to be rendered around the surface
type ViewerSurfaceAttachment = ComponentHook
```

The actual object rendering is done by the stack as described in section 1.7, the Renderer plugin is only providing a dynamically resized and centered surface for the stack to be drawn in. The surface is always kept square, even if the loaded BTF is not, to streamline and simplify texture coordinate handling.

1.9.8 PTMRenderer Plugin

The PTMRenderer Plugin is rendering the RGB and LRGB channel models. Here only the RGB is covered, as the principles for LRGB are the same. Most channel renderers will be split in two parts, one node for the rendering stack:

```
const coeffs = ['a0a1a2', 'a3a4a5']
// return a texture configuration array for the given coefficent
function mapper (btf: BTFFile, name: string) {
    return coeffs.map(c => {
        return btf.texForRender(name, c)
    })
}

// render a RGB object
const PTMRGB = Component<BaseNodeProps>(function PTMRGB (props) {
    let btf = props.appState.btf()
    return <Node
        // from ./ptmrgb.glsl
        shader={shaders.ptmrgb}
        // adaptive sizing if wanted
        width={props.width || btf.data.width}
```

```
        height={props.height || btf.data.height}
        uniforms={{
            // usually coming from the lightcontrol plugin, is [x:number, y:number, z:number]
            lightPosition: props.lightPos,
            // texture arrays
            texR: mapper(btf, 'R'),
            texG: mapper(btf, 'G'),
            texB: mapper(btf, 'B'),
            // retrieve the untyped formatExtra
            biases: (btf.data.formatExtra as PTMFormatMetadata).biases,
            scales: (btf.data.formatExtra as PTMFormatMetadata).scales,
        }} />
})
```

And one shader, implementing the lightening model described in ??.

```
precision highp float;
varying vec2 uv;
// higher and lower coefficents per color
uniform sampler2D texR[2];
uniform sampler2D texG[2];
uniform sampler2D texB[2];
uniform float biases[6];
uniform float scales[6];
uniform vec3 lightPosition;

float channelLum(sampler2D[2] coeffsTexs, vec3 toLight) {
    // would be unrolled by the GLSL compiler anyway
    float a0 = texture2D(coeffsTexs[0], uv).x;
    float a1 = texture2D(coeffsTexs[0], uv).y;
    float a2 = texture2D(coeffsTexs[0], uv).z;
    float a3 = texture2D(coeffsTexs[1], uv).x;
    float a4 = texture2D(coeffsTexs[1], uv).y;
    float a5 = texture2D(coeffsTexs[1], uv).z;

    a0 = (a0 * 255.0 - biases[0]) * scales[0];
    a1 = (a1 * 255.0 - biases[1]) * scales[1];
    a2 = (a2 * 255.0 - biases[2]) * scales[2];
    a3 = (a3 * 255.0 - biases[3]) * scales[3];
    a4 = (a4 * 255.0 - biases[4]) * scales[4];
    a5 = (a5 * 255.0 - biases[5]) * scales[5];

```

```
    float Lu = toLight.x;
    float Lv = toLight.y;

    float lum = (
        a0 * Lu * Lu +
        a1 * Lv * Lv +
        a2 * Lu * Lv +
        a3 * Lu +
        a4 * Lv +
        a5
    )/255.0;
    return lum;
}

void main() {
    // spotlight behaviour at the moment
    vec3 pointPos = vec3(0,0,0);
    vec3 toLight = normalize(lightPosition - pointPos);

    float R = channelLum(texR, toLight);
    float G = channelLum(texG, toLight);
    float B = channelLum(texB, toLight);

    gl_FragColor = vec4(R,G,B,1.0);
}
```

In addition to the default rendering mode the PTMRenderer plugin is also supporting a Normals rendering mode for LRGB ptms. See Figure 10 for a visual example. It is based on the proposal by MacDonald and Robson[16] and currently supports 4 extra rendering modes: X-Normal, Y-Normal, Z-Normal and False-Color-Normal. The first three render the respective value of the normal of the pixel as grey scale, the false color one maps X to R, Y to G and Z to B.

1.9.9 LightControl Plugin

The LightControl plugin is providing a visual way to control the position of the light source. The user interface is shown in Figure 11. The linear xy coordinates are transformed into hemispherical coordinates, which are then passed onto the current Base node from the Renderer plugin.

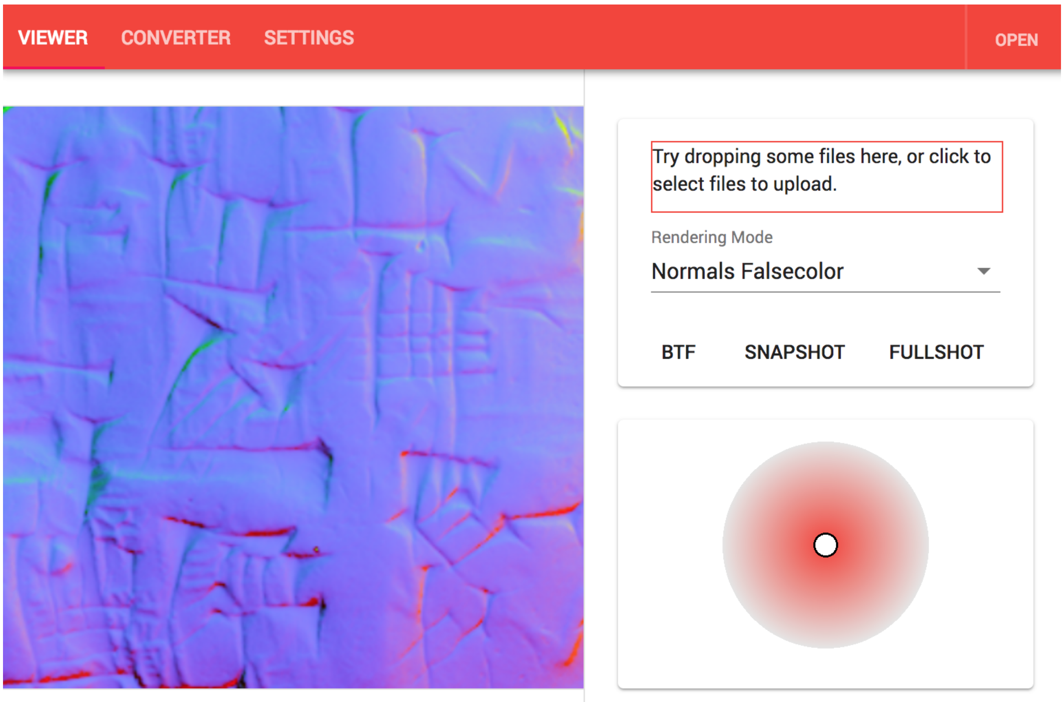

Figure 10: The Tablet ptm rendered with false color normals. The user can change the mode with the ‘Rendering Mode” drop down on the top right.

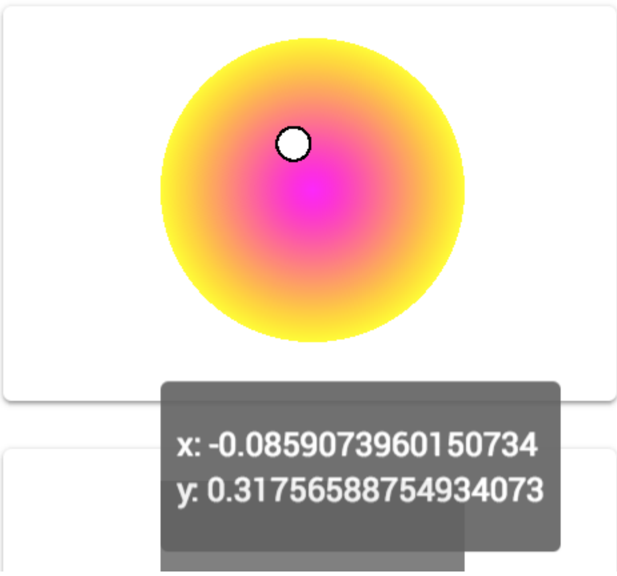

Figure 11: The user can drag around on the imaginary dome and change the lightening position.

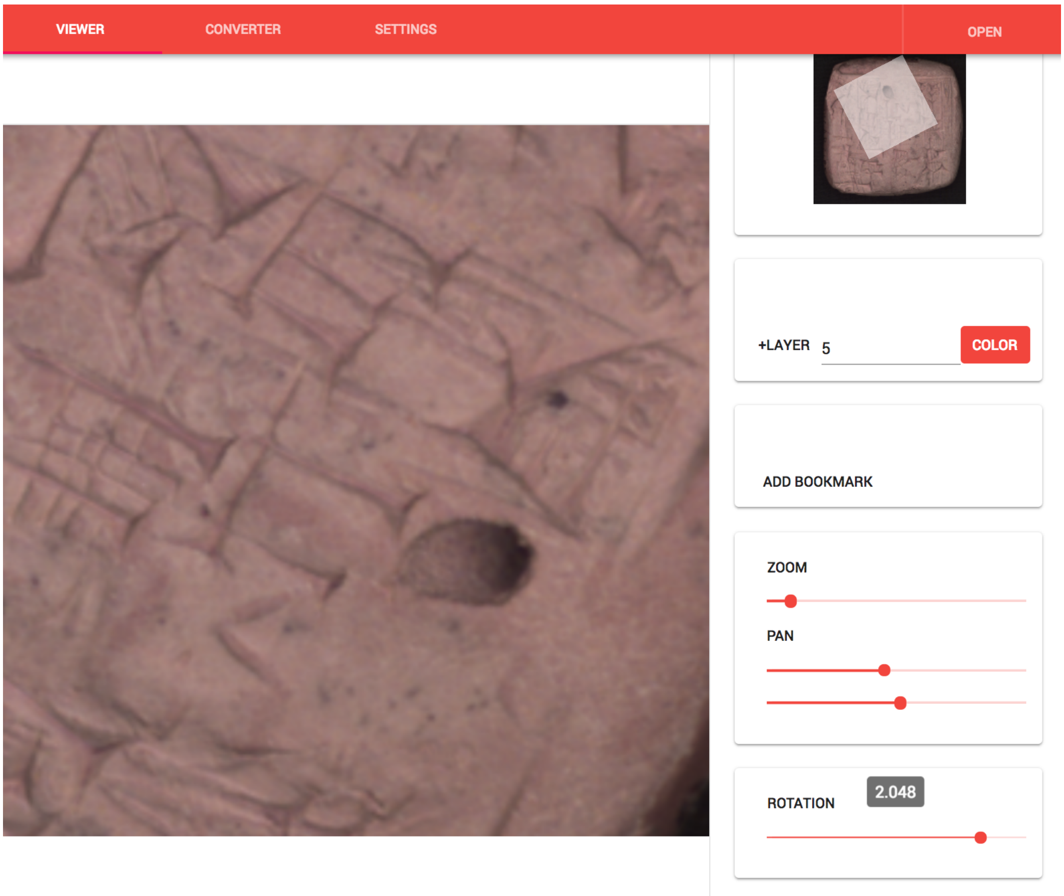

Figure 12: Rotated object. Note that the QuickPan views is also rotated.

1.9.10 Rotation Plugin

The Rotation plugin allows the rotation of the viewed object (and other lower rendering nodes). It does so by first centering the object inside a large square node of the maximal spanning length when rotated ($width * Math.cos(Math.PI/4) + height * Math.sin(Math.PI/4)$), so no part will be cut off for the rendering layers on top, and then applying a simple rotation matrix on the texture coordinates on the next node. An applied rotation is visible in Figure 12. Even though the whole renderer is rotated, the use of rotation slider will dynamically adapt the pan values to keep the previous center centered.

1.9.11 Zoom Plugin

The Zoom plugin is providing zoom and pan functionality (pan would not be necessary without zoom), an example is Figure 13.

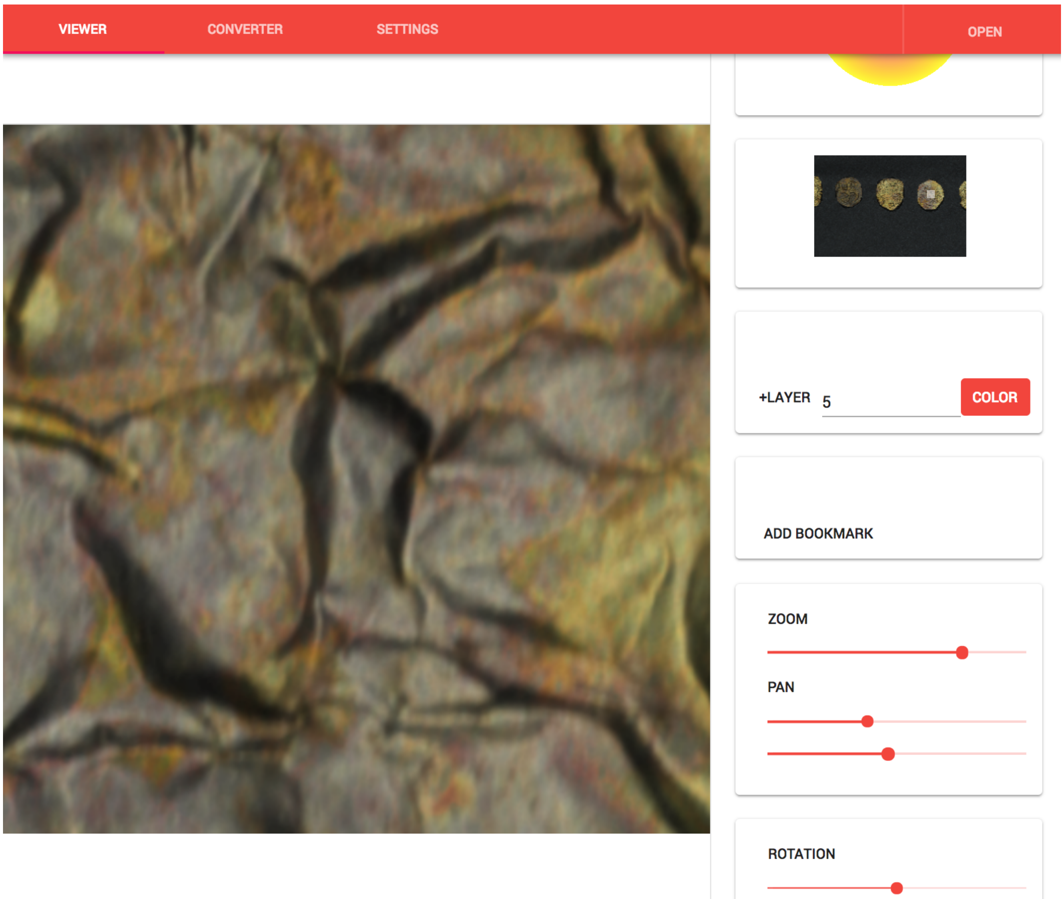

Figure 13: Applied zoom, note the small part of the actual object being shown. The shader architecture allows for seamless and performant zooming between nearest viewpoints and more removed ones.

1.9.12 QuickPan Plugin

The QuickPan plugin is responsible for rendering the small view box on the top right inside Figure 12 and Figure 13. It is doing so by rendering injecting a temporary node inside the rendering stack, waiting for an object's first render and then preserving that texture in a lower resolution, by setting the `width` and `height` properties of the nodes (compare the component in section 1.9.8). It then uses the RectRender component to render the semi-transparent rectangle of the current visible part over the captured texture. To do so, it is using the `inversePoint` function on `[0, 0]`, `[0,1]`, `[1,1]`, `[0,1]` to transform the bounds of the currently rendered (main) surface into texture coordinates of the object. With these coordinates a simple point-in-rectangle check can be done to highlight the desired area.

1.9.13 Paint Plugin

The Paint plugin adds the functionality to have overlays on the object to allow for annotations in a research context. All painting is happening as part of the renderer stack, with one node for each layer and one mixer node to combine these with the object. As the Paint plugin is interesting in terms of implementation, it is presented in more detail. The main Paint node:

```
/**
 * Actual painting node inside the render stack
 */
const PaintNode = Component(function PaintNode (props) {
    let btf = props.appState.btf()
    let width = btf.data.width
    let height = btf.data.height
    let brush = this.brushRadiusTex
    // just render the input texture if we got no other layers to put on top
    if (this.layers.length === 0)
        return <Node
            ref={this.handleRef('mixer')}
            width={width}
            height={height}
            key={this.key}
            shader={{
                frag: initShader,
            }}
            uniforms={{
                children: props.children,
            }} />
```

```
    else
        // return one mixer node, which stiches the underlying rendered object and the annotations together
        return <Node
            ref={this.handleRef('mixer')}
            width={width}
            height={height}
            key={this.key}
            onDraw={this.initialized ? null : this.onDraw}
            shader={{
                // dynamic shader for the current amount of layers
                frag: this.mixerShader(),
            }}
            uniforms={{
                children: props.children,
                layerVisibility: this.layers.map(l => l.visible),
                // convert mobx array to WebGL compatible one
                center: this.center.slice(0, 3),
                brushRadius: brush,
                showBrush: this.drawing === DrawingState.Hovering,
            }}
        >
            {// map all layers into the `layer` uniform of the mixer shader
                this.layers.map((layer, index) => {
                    // only change uniforms of currently drawn layer to not trigger redraws on stable layers
                    let drawThis = this.drawing === DrawingState.Drawing && this.activeLayer === index
                    return <Bus uniform={'layer'} key={`${layer.id}`} index={index} ><Node
                        // keep track of node refs for export
                        ref={this.handleRef(layer.id)}
                        width={width}
                        height={height}
                        shader={{
                            frag: this.initialized ? paintShader : initShader,
                        }}
```

```
                        clear={null}
                        uniforms={this.initialized ?
                            {
                                drawing: drawThis,
                                color: drawThis ? this.color.slice(0, 4) : [0, 0, 0, 0],
                                center: drawThis ? this.center.slice(0, 3) : [0, 0],
                                brushRadius: drawThis ? brush : 0,
                            } : {
                                // clear is null, so we initially just render the loaded texture
                                children: btf.annotationTexForRender(layer.id),
                            }} />
                    </Bus>
                })}
        </Node >
})
```

The shaders are comparatively simple. Paint first:

```
precision highp float;
varying vec2 uv;
uniform bool drawing;
uniform vec4 color;
uniform vec2 center;
uniform float brushRadius;
// the texture is permanent/not-cleared, if the shader discards, the old value is kept
void main() {
    if (drawing) {
        // only do changes if we are drawing currently
        vec2 d = uv - center;
        // paint if our point is near enough to the brush center
        if (length(d) < brushRadius) {
            gl_FragColor = color;
        } else {
            discard;
        }
```

```
    } else {
        discard;
    }
}
```

Then mixer, which is run through a string replacement first, as the WebGL compiler is not supporting loops with unfixed amounts of maximal iterations.

```
// Adapted shader to have fixed unrollable loops
mixerShader () {
    return mixerShader.replace(/\[X\]/gi, `[${this.layerCount}]`).replace('< layerCount', ` < ${this.layerCount}`)
}
```

Mixer source:

```
precision highp float;
varying vec2 uv;
uniform sampler2D children;
uniform bool layerVisibility[X];
uniform sampler2D layer[X];
uniform vec2 center;
uniform bool showBrush;
uniform float brushRadius;

void main() {
    vec4 base = texture2D(children, uv);
    // iterate over all layers
    for (int i=0; i < layerCount; i++) {
        if (layerVisibility[i]) {
            vec4 paint = texture2D(layer[i], uv);
            // and mix their color into the current color according to the layer's transparency
            base = mix(base, paint, paint.a);
            // the result should always be opaque
            base.a = 1.0;
        }
    }

    // preview brush rendering to visualize the brush size (and rendering lag)
    if (showBrush) {
        vec2 d = uv - center;
        if (length(d) < brushRadius) {
```

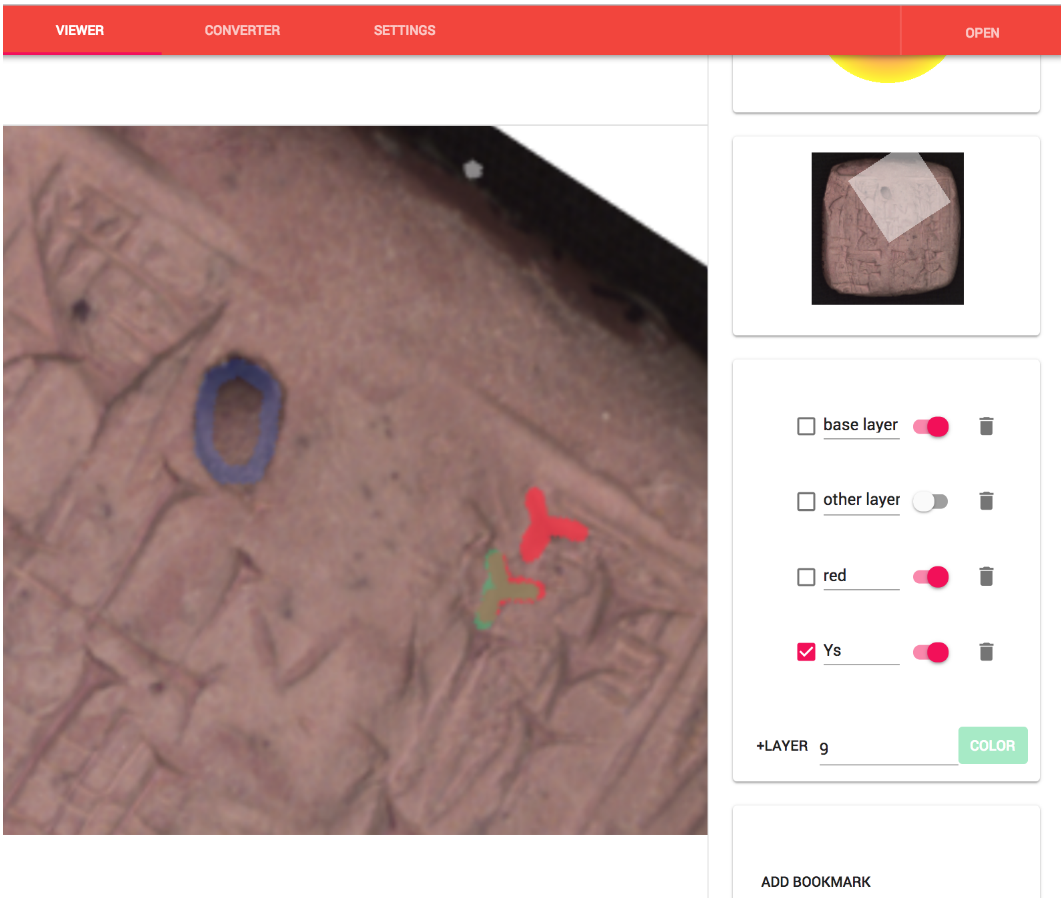

Figure 14: Paint plugin user interface. The check mark in front indicates the layer is currently drawn on. Names are freely changeable. The toggle toggles visibility of the layer. The trash icon deletes the corresponding layer. +Layer adds another layer on top. The number is the brush size. Clicking color shows a colour picker.

```
            base = mix(base, vec4(0.5, 0.5, 0.5, 0.5), 0.5);
        }
    }

    gl_FragColor = base;
}
```

The user interface is shown in Figure 14. Up to 15 layers are supported currently, as the max texture limit is usually 16.

1.9.14 Bookmarks Plugin

The Bookmarks plugins allows bookmarks to be set for light and view configurations. Bookmarks are controlled via the following hook:

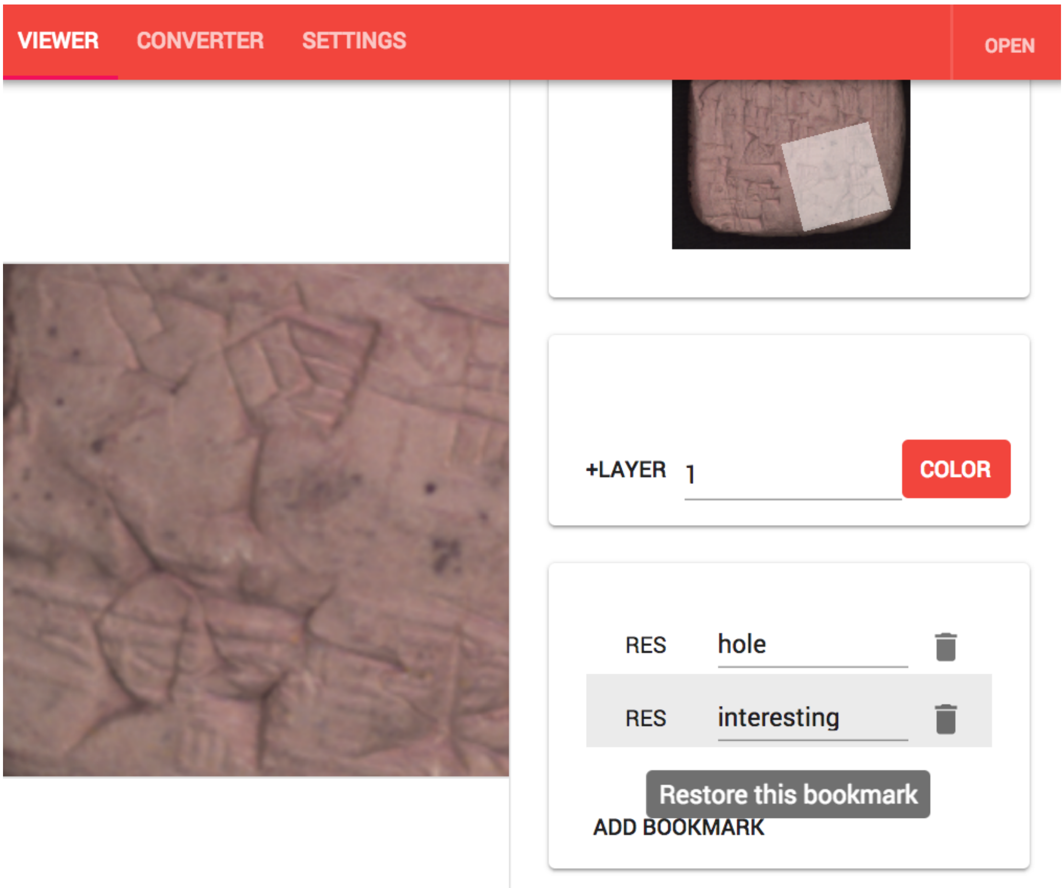

Figure 15: A click on 'Add Bookmark' will preserve the current light and camera positions and create an unnamed entry in the bookmark list. Clicking 'RES' restores this configuration. They are automatically exported when saving the current BTF.

```
type BookmarkSaver = {
    // key inside the bookmarks config
    key: string,
    // called when a new bookmark is created, any returned
     ↪ string or number combination is stored
    save: () => (string | number)[],
    // called when a bookmark ought to be restored, the saved
     ↪ combination is passed on
    restore: (values: (string | number)[]) => void,
}
```

1.9.15 Notes Plugin

The Notes plugin implements notes, which are attached to a specific point of the viewed object. Each notes consists of the parts: One indicator which is

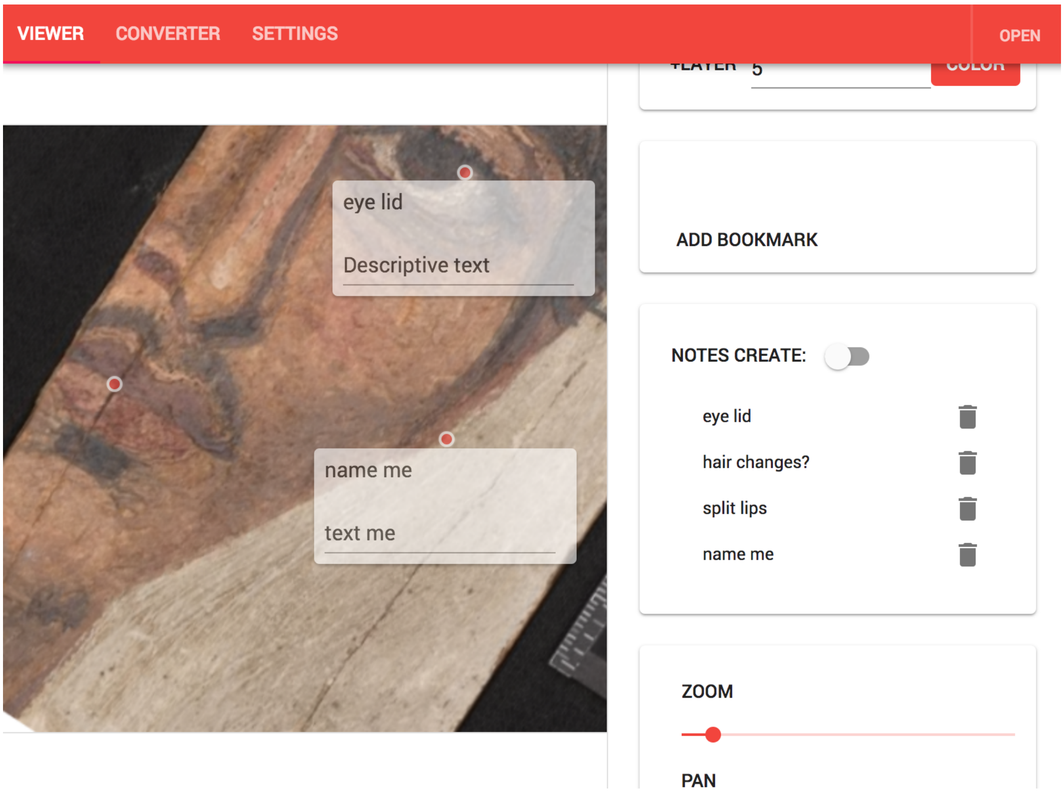

Figure 16: Notes are created by switching the ‘Notes Create:’ toggle and then clicking anywhere on the texture. The red and white dots indicate the presence of a note at that point, clicking it will toggle the display of the note. Clicking the note’s name on the right side, will center that note’s indicator.

linked to the texture and whose position is transformed like the texture and one popup, which calculates its absolute screen coordinates to show the note’s name and text. See Figure 16 for an example.

1.9.16 ImpExp Plugin

The ImpExp plugin is implementing the import and export of the current application state into/from a BTF file. For export the whole state tree is just exported as:

```
btf.oxrtiState = (this.appState as any).toJSON()
```

And for import the state is simply assigned as

```
for (let key in snapshot)
   (this.appState as any)[key] = snapshot[key]
```

The simpleness of this plugin is a big payoff for using mobx-state-tree and classy-mst. mobx-state-tree wil take the partial snapshosts and classy-mst will

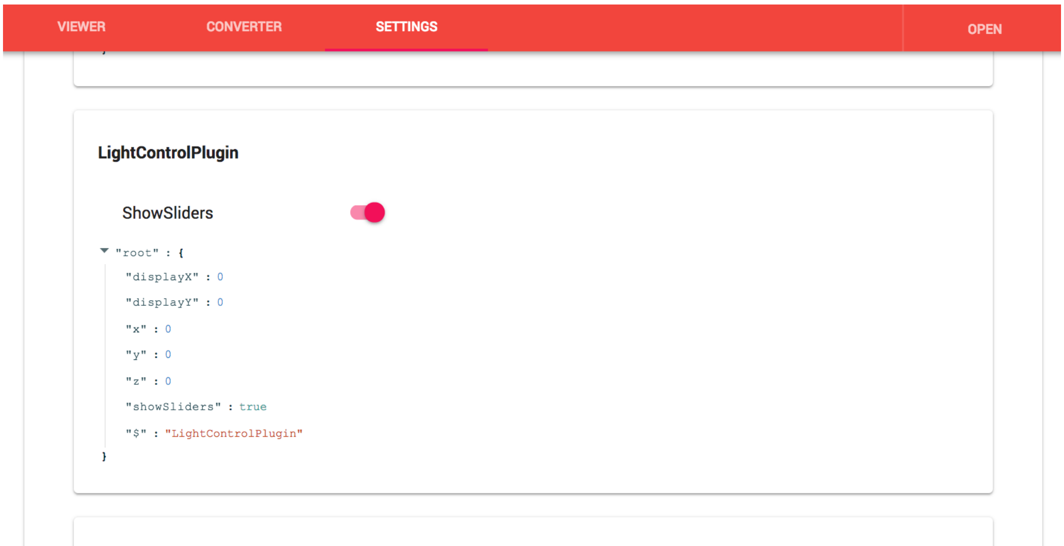

Figure 17: The LightControl plugin registers a toggle for enabling/disabling a slider based light control.

turn these into fresh instances of the respective plugins. All future plugins will be automatically supported, as long as they are using their models in the default way.

1.9.17 Settings Plugin

The Settings plugin provides the user with introspection into the currently loaded plugins and their configuration options. Plugins use the the SettingsConfig hook to register configuration values:

```
type SettingsConfig = ComponentHook & {
    title: string,
}
```

The Settings plugin registers a tab and will show the configuration components, see Figure 17.

1.10 Targets

The multiple targets are configured via two mechanisms: Webpack configuration files, which are configuring the compilation and bundling of the source code and other resources. And the *oxrti.plugins.json*, which has following form:

```
{
    "enabled": [
```

```
        "BasePlugin",
        "BaseThemePlugin",
        // ...
        "ZoomPlugin"
    ],
    "disabled": [
        "SingleViewPlugin",
        "UndoPlugin",
        "TestPlugin",
        // ...
    ]
}
```

The disabled block is not used within the implementation, but it is convenient for a configurator to see which plugins could be included. The loader is then only loading the configured plugins when starting the app.

1.10.1 Electron

Most of the electron compilation is controlled by the electron-webpack package, with additions in *webpack.renderer.additions.js* and *webpack.renderer.shared.js*. The main process entry point is *src/electron/index.tsx*, which contains no further modification at this point, but could be extended in the future for more native application features. The web process' entry point is *src/renderer/index.tsx*, which is installing the electron extensions and the referring to the Loader for the rest of the application startup. Compilation is started by executing `npm run-script electronbuild` on the command line, the resulting artifacts will be in the *dist* folder, in the MacOS use case it will be *dist/oxrti-*.dmg*. The development version can be started with `npm start` on the CLI.

1.10.2 Web

The pure web target is controlled by *webpack.config.js* and *webpack.renderer.shared.js*. `npm run-script startweb` will start the local development server listening on *http://localhost:3000*. `npm run-script build` will build it statically, with *index.html* and *dist/bundle.js* referring to the latest built versions. A zip containing both files with fixed relative links is also automatically generated under *dist/oxrti.zip*. This zip can be transferred to any computer and a modern web browser should be able to run the contained implementation.

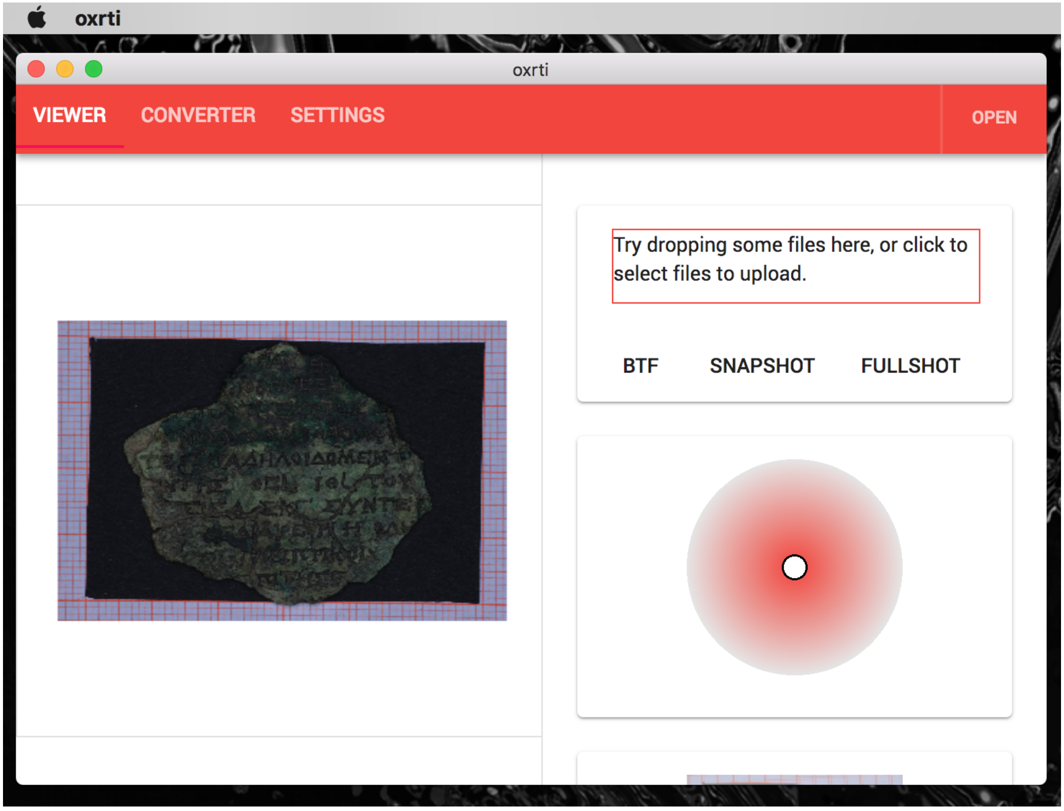

Figure 18: The bundled electron app. Later versions will add some kind of native menu.

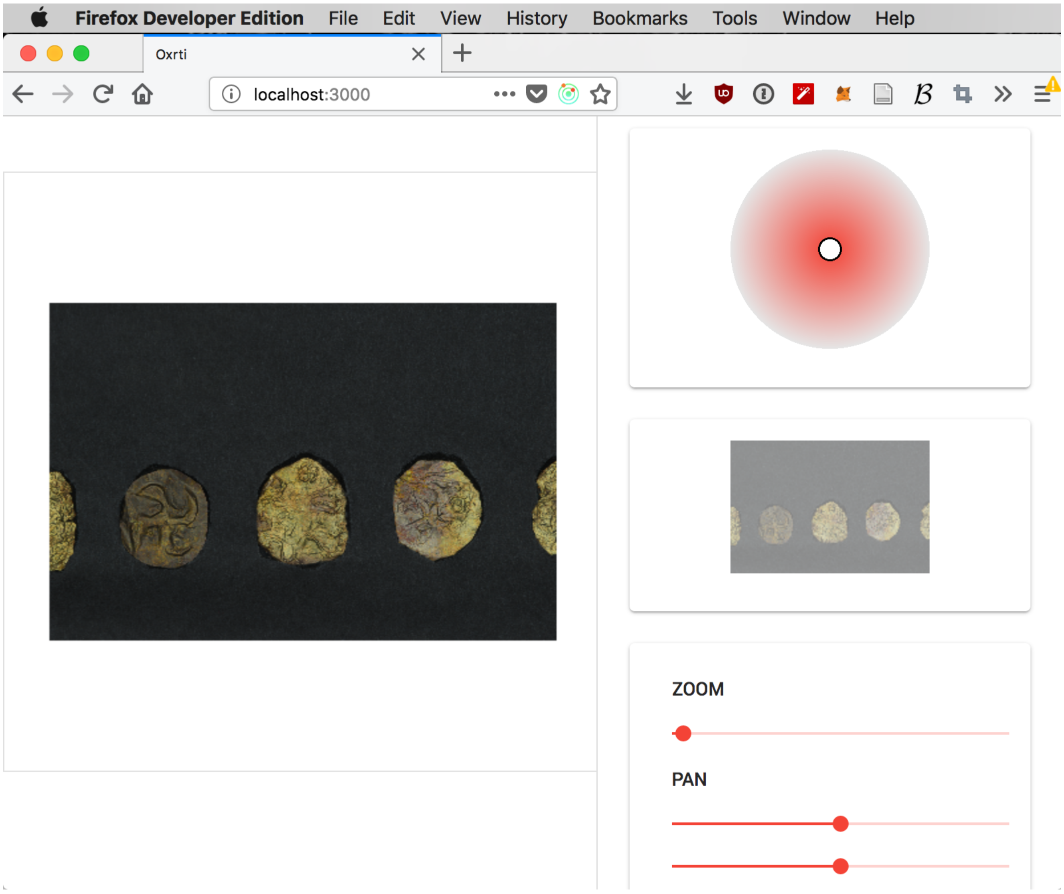

Figure 19: Embedded web version, with SingleView plugin and without Converter, Bookmarks and Paint.

1.10.3 Hosted

The build scripts are also generating a hosted version on each commit. On push the git repository[12] is automatically feeding this to an Azure Functions instance, reachable at `https://oxrtimaster.azurewebsites.net/api/azurestatic`. This process is controlled by *webpack.functions.js* and the files in *azurejs* and *azurestatic*, which are basically creating two endpoints on the azure site, each hosting the compiled *.js* file or *.html* file.

1.10.4 Embeddable

There is currently no provisioning for a more automated handling of multiple *oxrti.plugin.json* files, so it needs to be exchanged manually before building. A proposed embedded configuration for e.g. object galleries is provided by *oxrti.plugins.embedded.json*, a resulting version is depicted in Figure 19.

1.10.5 Mobile Phones

Mobile phones are supported (see Figure 20), but currently might have problems with the distinction between *touch* and *click* events and currently have no specialized layout.

2 Results

This sections summarises the results of the project and adds some quantitative analysis of the implementation. The feature set is fully reflected inside the implementation section (or the table of contents), so no additional summarization is done here.

2.0.6 Performance

The currently most used RTI viewer is the Cultural Heritage Imaging one's[4], so the performance comparisons are done against that implementation. The data is shown in Table 1. The BTF files are about 50% the size of the ptm files, with increasing compression benefits the larger the ptm file is. The same pattern is evident in the load time as Figure 21 shows. The bigger the files are, the more profitable is the shader implementation.

File	Pixel	Conv	Load	Load RTI	ptm size	btf size
AK1A_LRGB	3008x2000	4906	1713	11800	54.1	30
Coin_LRGB	7360x4912	24674	8224	59200	325.4	136
Mummy_RGB	583x1000	1769	627	5200	10.5	5.6
tablet2_LRGB	512x512	732	462	1600	2.4	1.2
Warrior_RGB	2299x3200	10590	2184	25500	132.4	50.2

Table 1: Performance Comparison

File names refer to the files distributed as [13]. *Pixel* are the dimensions as width x height. *Conv* is the time taken by the implementation to convert the ptm file to the btf format. *Load* is the time taken by the oxrti implementation from receiving the btf file to do a complete render. *Load RTI* is the approximate time taken by the RTIViewer from opening the file to finishing the loading (time stopped by screenrecording, as no performance measurements are provided, time for processing of mipmaps and normal maps excluded, as the oxrti implementation is not doing these steps, time to full first render would be even longer). *ptm size* is the file size in megabytes. *btf size* is the file size of a plain BTF file (no extra oxrti state and/or layers).

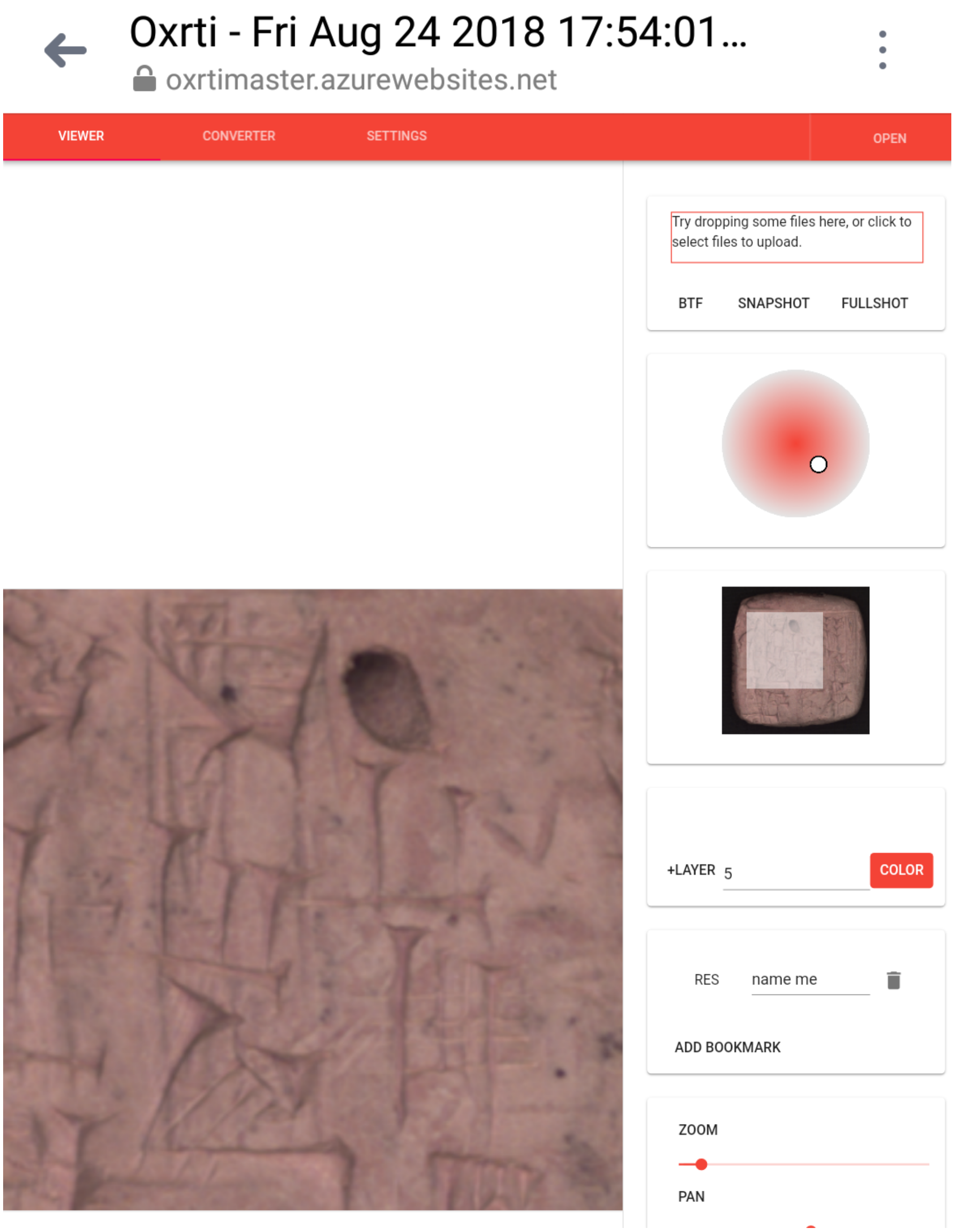

Figure 20: Implementation running on a Huawei P10 mobile phone with the Chrome browser. Opening a BTF is done through clicking the upload area.

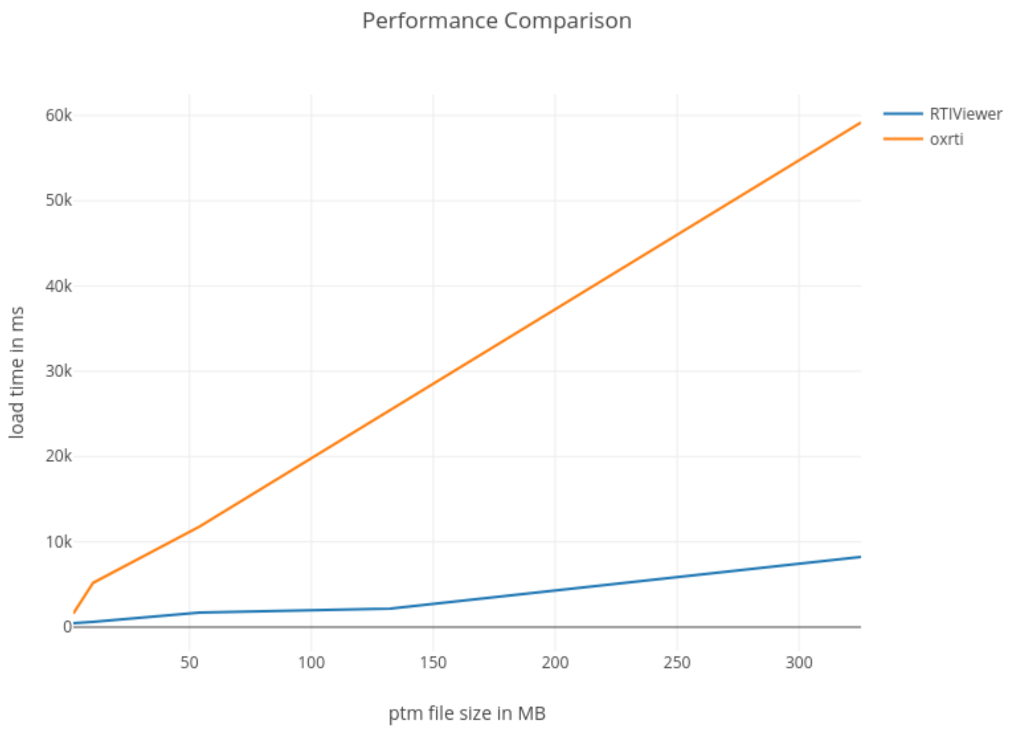

Figure 21: Load time comparison.

2.1 Accuracy

The next thing to verify is the accuracy, is the implemented rendering actually the 'right' imagine. To quantify that, the Root Mean Square Error as presented by Happa and Gogosio[11] was picked, as it the compared files have rendering

File	RMSE(R)	RMSE(G)	RMSE(B)	SUM
tablet2_LRGB	0.649	0.677	0.674	2.000
Warrior_RGB	0.627	0.612	0.610	1.849

Table 2: Table to test captions and labels

2.2 Testing

Todo Text:
Testing

Todo Text:
Shader Interpolation

Todo Text:
Image comparison

2.3 Rollouts and Deployments

Todo Text:
Rollout

Todo Text:
Non-Tech deployment

o

This is the specification of the BTF fileformat, as of version 1.0 on August 26, 2018. It was developed co-supervisor Stefano Gogioso, with input from and extension by the author of the enclosing thesis in relation to the oxrti viewer.

A BTF File Format

This section describes the BTF file format. The aim of this file format is to provide a generic container for BTF data to be specified using a variety of common formats. Files shall have the `.btf.zip` extension.

A.1 File Structure

A BTF file is a ZIP file containing the following:

- A **manifest** file in JSON format, named `manifest.json`. The manifest contains all information about the BRDF/BSDF model being used, including the names for the available **channels** (e.g. `R`, `G` and `B` for the 3-channel RGB), the names of the necessary **coefficients** (e.g. bi-quadratic coefficients) and the **image file format** for each channel.
- A single folder named `data`, with sub-folders having names in 1-to-1 correspondence with the channels specified in the manifest.
- Within each channel folder, greyscale image files having names in 1-to-1 correspondence with the coefficients specified in the manifest, each in the image file format specified in the manifest for the corresponding channel. For example, if one is working with RGB format (3-channels named `R`, `G` and `B`) in the PTM model (five coefficients `a2`, `b2`, `a1`, `b1` and `c`,specifying a bi-quadratic) using 16-bit greyscale bitmaps, the file `/data/B/a2.bmp` is the texture encoding the `a2` coefficient for the blue channel of each point in texture space.
- The datafiles are all in reversed scanline order (meaning from bottom to top), to keep aligned with the original PTM format and allow easier loading into WebGL.

In case of usage with the oxrti viewer, following files can be present in addition to those mentioned above:

A.2 Manifest

The manifest for the BTF file format is a JSON file with root dictionary. The `root` element has two mandatory child elements: one named `data`, and one named `name` with the option of additional child elements (with different names) left open to future extensions of the format.

- The `name` element is a string with a name of the contained object.
- The `data` element has for entries, named `width`, `height`, `channels` and `channel-model`. The `width` and `height` attributes have values in the positive integers describing the dimensions of the BTDF. The `channel-model` attribute has value a non-empty alphanumeric string uniquely identifying the BRDF/BSDF colour model used by the BTF file (see Options section below). The `channels` element has an arbitrary amout of named `channel` entries, according to the `channel-model`.
- Additionally the `data` element has one untyped entry named `formatExtra`, where format implementation specific data can be stored.
- Each `channel` has an `coefficents` child consisting of an arbitrary number of `coefficent` entries, as well as one `coefficient-model` attribute. The `coefficient-model` attribute has value a non-empty alphanumeric string uniquely identifying the BRDF/BSDF approximation model used by the BTF file (see Options section below).
- Each `coefficient` element has one attribute: `format`. The `format` attribute has value a non-empty alphanumeric string uniquely identifying the image file format used to store the channel values (see Options section below).

A.3 Textures

Each image file `/data/CHAN/COEFF.EXT` has the same dimensions specified by the `width` and `height` attributes of the `data` element in the manifest, and is encoded in the greyscale image file format specified by the `format` attribute of the unique `coefficient` element with attribute `name` taking the value `COEFF` (the extension `.EXT` is ignored). The colour value of a pixel `(u,v)` in the image is the value for coefficient `COEFF` of channel `CHAN` in the BRDF/BSDF for point `(u,v)`, according to the model jointly specified by the values of the attribute `model` for element `channels` (colour model) and the attribute `model` for element `coefficients` (approximation model).

A.4 Options

At present, the following values are defined for attribute `channel-model` of element `channels`.

- `RGB`: the 3-channel RGB colour model, with channels named `R`, `G` and `B`. This colour model is currently under implementation.
- `LRGB`: the 4-channel LRGB colour model, with channels named `L`, `R`, `G` and `B`. This colour model is currently under implementation.
- `SPECTRAL`: the spectral radiance model, with an arbitrary non-zero number of channels named either all by wavelength (format `---nm`, with `---` an arbitrary non-zero number) or all by frequency format `---Hz`, with `---` an arbitrary non-zero number. This colour model is planned for future implementation.

At present, the following values are defined for attribute `model` of element coefficients, where the ending character `*` is to be replaced by an arbitrary number greater than or equal to 1.

- `flat`: flat approximation model (no dependence on light position). This approximation model is currently under implementation.
- `RTIpoly*`: order-`*` polynomial approximation model for RTI (single view-point BRDF). This approximation model is currently under implementation.
- `RTIharmonic*`: order-`*` hemispherical harmonic approximation model for RTI (single view-point BRDF). This approximation model is currently under implementation.
- `BRDFpoly*`: order-`*` polynomial approximation model for BRDFs. This approximation model is planned for future implementation.
- `BRDFharmonic*`: order-`*` hemispherical harmonic approximation model BRDFs. This approximation model is planned for future implementation.
- `BSDFpoly*`: order-`*` polynomial approximation model for BSDFs. This approximation model is planned for future implementation.
- `BSDFharmonic*`: order-`*` spherical harmonic approximation model for BSDFs. This approximation model is planned for future implementation.

At present, the following values are defined for attribute `format` of elements tagged `coefficient`, where the ending character `*` is the bit-depth, to be replaced by an allowed positive multiple of 8.

- `BMP*`: greyscale BMP file format with the specified bit-depth (8, 16, 24 or 32). Support for this format is currently under implementation.

- `PNG*`: PNG file format encoding the specified bit-depth (8, 16, 24, 32, 48 or 64). Support for this format is currently under implementation. Different PNG colour options are used to support different bit-depths:
- `Grayscale` with 8-bit/channel to encode 8-bit bit-depth.
- `Grayscale` with 16-bit/channel to encode 16-bit bit-depth.
- `Truecolor` with 8-bit/channel to encode 24-bit bit-depth.
- `Truecolor and alpha` with 8-bit/channel to encode 32-bit bit-depth.
- `Truecolor` with 16-bit/channel to encode 48-bit bit-depth.
- `Truecolor and alpha` with 16-bit/channel to encode 64-bit bit-depth.

References

[1] *Alternative syntax madness · Issue #487 · mobxjs/mobx-state-tree*. GitHub. URL: https://github.com/mobxjs/mobx-state-tree/issues/487 (visited on 08/21/2018).

[2] *Classes · TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/classes.html (visited on 08/20/2018).

[3] *classy-mst: ES6-like syntax for mobx-state-tree*. Aug. 11, 2018. URL: https://github.com/charto/classy-mst (visited on 08/13/2018).

[4] *Cultural Heritage Imaging | View: RTIViewer Download*. URL: http://culturalheritageimaging.org/What_We_Offer/Downloads/View/ (visited on 08/26/2018).

[5] *Decorators · TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/decorators.html (visited on 08/20/2018).

[6] *electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*. Aug. 21, 2018. URL: https://github.com/electron/electron (visited on 08/21/2018).

[7] *electron-userland/electron-webpack: Scripts and configurations to compile Electron applications using webpack*. URL: https://github.com/electron-userland/electron-webpack (visited on 08/21/2018).

[8] *Generics · TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/generics.html (visited on 08/20/2018).

[9] *gl-react cookbook*. URL: https://gl-react-cookbook.surge.sh/hellogl (visited on 08/21/2018).

[10] *GNU Emacs - GNU Project*. URL: https://www.gnu.org/software/emacs/ (visited on 08/17/2018).

[11] Stefano Gogioso and Jassim Happa. "PBR: Image Synthesis Validation". Physical Based Rendering course, Oxford University., Nov. 24, 2017.

[12] Johannes Goslar. *oxrti: Reflectance Transformation Imaging Toolset*. Aug. 15, 2018. URL: https://github.com/ksjogo/oxrti (visited on 08/17/2018).

[13] Johannes Goslar. *oxrti_data: public data test files for the oxrti project*. Aug. 26, 2018. URL: https://github.com/ksjogo/oxrti_data (visited on 08/26/2018).

[14] Takahiro Ethan Ikeuchi. *React Stateless Functional Component with TypeScript*. Medium. Apr. 5, 2017. URL: https://medium.com/@ethan_ikt/react-stateless-functional-component-with-typescript-ce5043466011 (visited on 08/20/2018).

[15] *JSZip*. URL: https://stuk.github.io/jszip/ (visited on 08/21/2018).

[16] Lindsay MacDonald and Stuart Robson. "Polynomial texture mapping and 3d representations". In: *nternational Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 38* 5 (2010), p. 6.

[17] *material-ui: React components that implement Google's Material Design.* Aug. 21, 2018. URL: https://github.com/mui-org/material-ui (visited on 08/21/2018).
[18] *mobx: Simple, scalable state management.* Aug. 13, 2018. URL: https://github.com/mobxjs/mobx (visited on 08/13/2018).
[19] *mobx-state-tree: Model Driven State Management.* Aug. 20, 2018. URL: https://github.com/mobxjs/mobx-state-tree (visited on 08/21/2018).
[20] *React - A JavaScript library for building user interfaces.* URL: https://reactjs.org/index.html (visited on 08/13/2018).
[21] *Redux DevTools.* URL: https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd (visited on 08/22/2018).
[22] Gaëtan Renaudeau. *gl-react – React library to write and compose WebGL shaders.* Aug. 13, 2018. URL: https://github.com/gre/gl-react (visited on 08/13/2018).
[23] Arian Stolwijk. *pngjs: Pure JavaScript PNG decoder.* July 20, 2018. URL: https://github.com/arian/pngjs (visited on 08/21/2018).
[24] *The WebGL API: 2D and 3D graphics for the web.* MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (visited on 08/21/2018).
[25] *TSLint.* URL: https://palantir.github.io/tslint/ (visited on 08/17/2018).
[26] *TSLint - Visual Studio Marketplace.* URL: https://marketplace.visualstudio.com/items?itemName=eg2.tslint (visited on 08/17/2018).
[27] *TypeScript is a superset of JavaScript that compiles to clean JavaScript output.* Aug. 13, 2018. URL: https://github.com/Microsoft/TypeScript (visited on 08/13/2018).
[28] *Visual Studio Code - Code Editing. Redefined.* URL: http://code.visualstudio.com/ (visited on 08/17/2018).
[29] *WebGL Specification.* URL: https://www.khronos.org/registry/webgl/specs/1.0/ (visited on 08/21/2018).
[30] *WebGL Stats.* URL: http://webglstats.com/ (visited on 08/21/2018).
[31] *WebGL Stats Texture Units.* URL: http://webglstats.com/webgl/parameter/MAX_TEXTURE_IMAGE_UNITS (visited on 08/21/2018).
[32] *WebGL tutorial.* MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial (visited on 08/21/2018).
[33] *webpack/webpack: A bundler for javascript and friends. Packs many modules into a few bundled assets. Code Splitting allows to load parts for the application on demand. Through "loaders," modules can be CommonJs, AMD, ES6 modules, CSS, Images, JSON, Coffeescript, LESS, ... and your custom stuff.* URL: https://github.com/webpack/webpack (visited on 08/21/2018).