, titleaddon = GitHub, urldate = 2018-08-21, langid = english, file = Snapshot:/Users/dkd-goslar/Zotero/storage/N7QYHTKT/487.html:text/html

*Todo list

Todo Text:
Background in Applications 1
Todo Text:
Background in Computer Science 1
Todo Text:
Related work intro 1
Todo Text:
Workflow Comparisions 1
Todo Text:
File formats comparison 1
Todo Diagramm:
Size tables/graphes of ptm/rti/btf(.zip) 1
Todo Text:
Streaming architectures 1
Todo Text:
Viewer Comparision 2
Todo Text:
No extensible architecture 2
Todo Text:
No real open source (email before or one file sources) 2
Todo Text:
Camera Theory 2
Todo Text:
Requirements Analysis, informal discussion 2
Todo Text:
Architecture Picks 2
Todo Text:
State-Driven 4
Todo Text:
Plugins 4
Todo Text:
Rendering Stack 4
Todo Diagramm:
Workflow comparison 4
Todo Text:
File import/export 4

Todo Text:
Novelties Design 5
Todo Text:
state management 21
Todo Text:
state import/export 21
Todo Diagramm:
redux 21
Todo Diagramm:
mobx actions 21
Todo Text:
base rendering nodes 21
Todo Diagramm:
stacked components 21
Todo Diagramm:
effects 21
Todo Text:
Texture Loading 21
Todo Text:
Plugins API 23
Todo Text:
Base Plugin 23
Todo Text:
Basetheme Plugin 23
Todo Text:
TabView Plugin 24
Todo Text:
SingleView Plugin 25
Todo Text:
Converter Plugin 25
Todo Text:
PTMConverter Plugin 25
Todo Text:
Renderer Plugin 25
Todo Text:
Base Node 25
Todo Text:
WebGL texture packing 25
Todo Text:
PTM Renderer Plugin 26

Todo Text:
Dynamic Shaders 26
Todo Text:
RGB vs LRGB 26
Todo Text:
Light Control Plugin 26
Todo Text:
Rotation Plugin 26
Todo Text:
Zoom Plugin 26
Todo Text:
Zoom Plugin 26
Todo Text:
Zoom Plugin 27
Todo Text:
Automatic Import Export 27
Todo Text:
Other related graphics 27
Todo Text:
Applications 27
Todo Text:
Standalone Website 27
Todo Text:
Embeddable 27
Todo Text:
Electron App deliverable 27
Todo Text:
Featureset Comparison 28
Todo Diagramm:
Screeshots 28
Todo Text:
Performance 28
Todo Text:
Testing 28
Todo Text:
Shader Interpolation 28
Todo Text:
Image comparison 28
Todo Text:
Rollout 28

Todo Text:
Non-Tech deployment 28
Todo Text:
Community Onboarding 29
Todo Text:
Novelties results 29
Todo Text:
Future Work 29
Todo Text:
Conclusion 29

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.

Acknowledgements

Contents

1 Introduction **1**
- 1.1 Background in Applications . . . 1
- 1.2 Background in Computer Science . . . 1

2 Related Work **1**
- 2.1 RTI Theory and Workflows . . . 1
- 2.2 Fileformats . . . 1
- 2.3 RTI Viewers . . . 2
- 2.4 Camera Theory . . . 2

3 Methodology **2**
- 3.1 Requirements . . . 2
- 3.2 Architectural Design . . . 2

4 Requirements and Design **3**
- 4.1 Requirements . . . 3
- 4.2 State-Driven . . . 4
- 4.3 Plugins . . . 4
- 4.4 Rendering Stack . . . 4
- 4.5 Workflow . . . 4
- 4.6 Fileformat . . . 5
- 4.7 Novelties . . . 5

5 Implementation **5**
- 5.1 Overview . . . 5
- 5.2 Libraries . . . 5
- 5.3 Overall Architecture . . . 16
- 5.4 BTF File Format . . . 17
  - 5.4.1 File Structure . . . 18
  - 5.4.2 Manifest . . . 18
  - 5.4.3 Textures . . . 19
  - 5.4.4 Options . . . 19
- 5.5 Texture Loader . . . 21
- 5.6 State Management . . . 21
- 5.7 Renderer Stack . . . 21
  - 5.7.1 Texture Loading . . . 21
- 5.8 Hooks . . . 21
- 5.9 Plugins . . . 23
  - 5.9.1 Base Plugin . . . 23

5.9.2 BaseTheme Plugin . . . 23
5.9.3 RedTheme Plugin . . . 24
5.9.4 TabView Plugin . . . 24
5.9.5 SingleView Plugin . . . 25
5.9.6 Converter Plugin . . . 25
5.9.7 PTMConverter Plugin . . . 25
5.9.8 Renderer Plugin . . . 25
5.9.9 PTM Renderer Plugin . . . 26
5.9.10 Light Control Plugin . . . 26
5.9.11 Rotation Plugin . . . 26
5.9.12 Zoom Plugin . . . 26
5.9.13 QuickPan Plugin . . . 26
5.9.14 Paint Plugin . . . 27
5.9.15 Import Export Plugin . . . 27
5.10 Applications . . . 27
5.10.1 Standalone Website . . . 27
5.10.2 Embeddable . . . 27
5.10.3 Electron . . . 27

6 Results **28**
6.1 Featureset . . . 28
6.2 Performance . . . 28
6.3 Testing . . . 28
6.4 Rollouts and Deployments . . . 28

7 Discussion **29**
7.1 Community Onboarding . . . 29
7.2 Novelties . . . 29
7.3 Future Work . . . 29
7.3.1 WebGL 2 . . . 29

8 Conclusion **29**

List of Figures

1 MobX Flow . . . 9
2 MoWebGL compability . . . 11
3 Mogl-react-cookbook example . . . 13
4 MoZoom Component . . . 16

List of Tables

1 Introduction

1.1 Background in Applications

Todo Text:
Background in Applications

1.2 Background in Computer Science

Todo Text:
Background in Computer Science

2 Related Work

Todo Text:
Related work intro

2.1 RTI Theory and Workflows

Todo Text:
Workflow Comparisions

2.2 Fileformats

The most comprehensive overview on the current state of the art is done by the American library of congress as part of its Digital preservation effort, with the sections on the ptm[4] and rti[5] formats. The current PTM specification by Malzbender and Gelb[14].

Todo Text:
File formats comparison

Todo Diagramm:
Size tables/graphes of ptm/rti/btf(.zip)

Todo Text:
Streaming architectures

2.3 RTI Viewers

Todo Text:
Viewer Comparision

Todo Text:
No extensible architecture

Todo Text:
No real open source (email before or one file sources)

2.4 Camera Theory

Todo Text:
Camera Theory

3 Methodology

Exploratory piece of work

3.1 Requirements

Todo Text:
Requirements Analysis, informal discussion

3.2 Architectural Design

Todo Text:
Architecture Picks

4 Requirements and Design

4.1 Requirements

Distilling these, I arrived at the following functional requirements, which are logically grouped into fileformat/support and viewer.

For the fileformat:

1. Support for the PTM[4] fileformat.
2. Support for the RTI[5] fileformat.
3. Conversion of the formats above into a unified format.
4. Extended metadata support.
5. Support for high resolutions.
6. Support for higher bitdepths per pixel than the 8 of PTM/RTI.
7. Easy exchange between multiple researchers.

For the viewer component:

8. Runnable on all major operating systems and/or web browsers.
9. Lightning Controls.
10. Quick navigation functionality.
11. Annotations.
12. Overlays.

Continuing the enumeration of the functional requirements, following non-functional requirements were extracted:

13. Free Software, the implementation should be available for everyone to change and distribute.
14. Easy on-boarding of new developers, either scientists in a research context or students in an education context.
15. Good developer experience.
16. Adequate performance, at least keeping up with current implementations.
17. Easy installation for researchers.

18. “Web”-Based.
19. Instant reactiveness.
20. Reasonable file sizes for instant transfer/viewing.
21. Preservable software and BTF files.

4.2 State-Driven

Todo Text:
State-Driven

4.3 Plugins

Todo Text:
Plugins

4.4 Rendering Stack

Todo Text:
Rendering Stack

4.5 Workflow

Todo Diagramm:
Workflow comparison

json

Todo Text:
File import/export

4.6 Fileformat

4.7 Novelties

Todo Text:
Novelties Design

5 Implementation

5.1 Overview

This section explains the current implementation of the developed tool set, it is primarily targeted to fulfill the dissertation's requirements. But is also aiming to be helpful for users wanting to understand the underlying systems and prepare them for potentially joining the development effort. Abridged code extracts are used as of their state for thesis submission, while the main principles will hold, later readers are asked to please consult the actual source code if any discrepancies arise or reexport the document. First the main libraries are shortly explained in their relevance to the program, second the largely abstract plugin architecture is shown, third 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`.

5.2 Libraries

TypeScript

The official header line of TypeScript show some points 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."[24] Which fits requirements 13, 8. 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 run and a quite improved developer experience (requirement 15) in the long run. With the full typed hook system (compare section 5.8) 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 setup in a way to fully embrace editor tooling, Visual Studio Code[25] and Emacs[11] 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[22] plugin[23] 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[6] and Generics[9], 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"[18], 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 14 and 15 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 16 and 19) as the necessary changes can be track and components be updated selectively.

```
// a class represents a single component
class Timer extends React.Component {
```

```
3     // the parent component can pass on props to it
4     constructor(props) {
5       super(props);
6       this.state = { seconds: 0 };
7     }
8
9     tick() {
10      //  the state is updated and the component is
          ↪  automatically rerendered
11      this.setState(prevState => ({
12        seconds: prevState.seconds + 1
13      }));
14    }
15
16    // called after the component was created/added to the
      ↪  browser window
17    componentDidMount() {
18      this.interval = setInterval(() => this.tick(), 1000);
19    }
20
21    // called before the component will be deleted/removed from
      ↪  the browser window
22    componentWillUnmount() {
23      clearInterval(this.interval);
24    }
25
26    // the actual rendering code
27    // html can be directly embedded into react components
28    // {} blocks will be evaluated when the render method is
      ↪  called
29    // which will happen any time the props or its internal
      ↪  states updates
30    render() {
31      return (
32        <div>
33          Seconds: {this.state.seconds}
34        </div>
35      );
36    }
37  }
38
```

```
// 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'[12]). 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"[16]. An introducing overview is shown in Figure 1. Broadly speaking MobX introduces observable objects. Instead of mentioned 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"[17] 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:

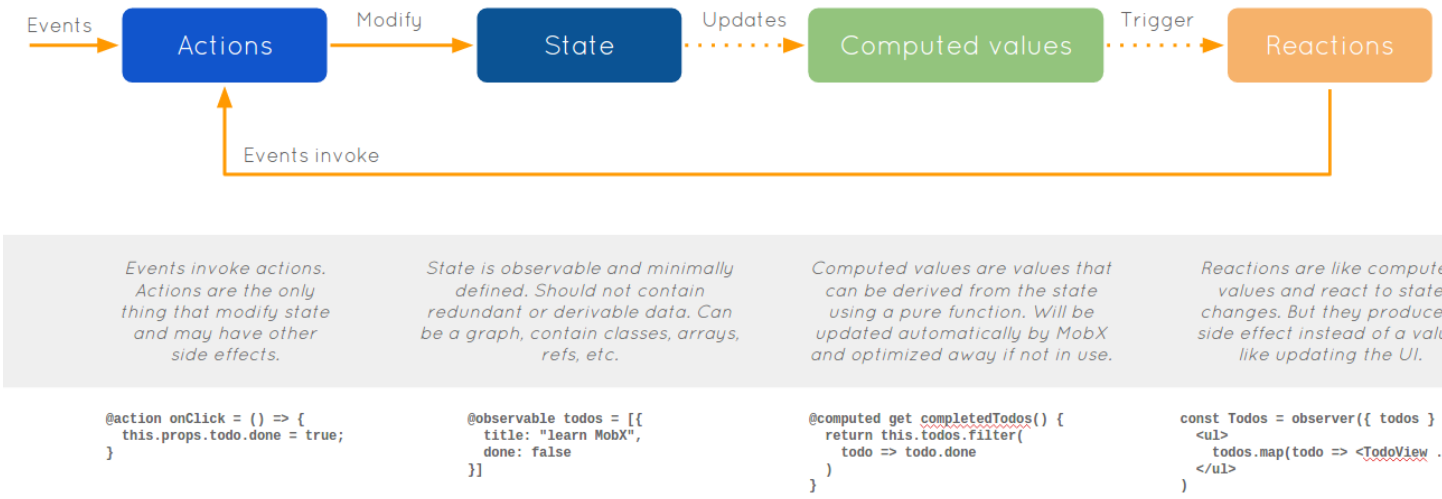


Figure 1: Taken from Weststrate[16]. Actions in the oxrti context are most often initially user actions, which are then calling into plugins to change the state. The state is mostly encapsulated on a plugin basis with usage of the mobx-state-tree library, which also encapsulates most computed values. Reactions are most often the previously discussed React components.

```
// 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 14.

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]:

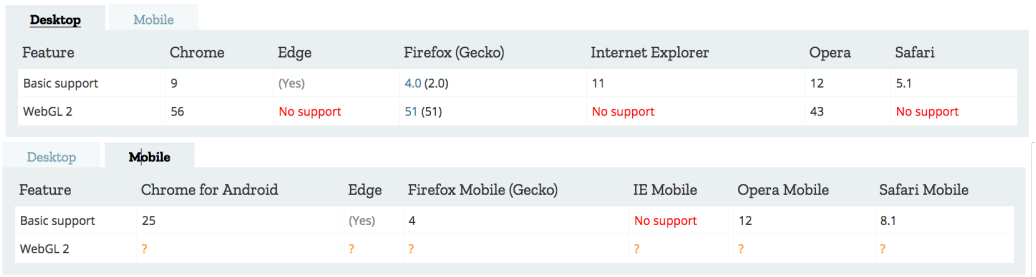
```
const TodoData = types.model({
    title: types.string,
    done: false

});

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

```

```
14 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."[29] 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 5.7. While preserving compatibility and requirement 8 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[27].



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[21].

Potential improvements when WebGL 2 is more widely supported or in conditional plugins are discussed in section 7.3.1. 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[28], whereas the standard OpenGL implementations are likely to have a limit of 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[26] and the the texture loader (section 5.5), 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.

"Implement complex effects by composing React components."[19] is the main use of the gl-react library. A minimal component, adapted from the gl-react-cookbook looks like[19]:

```
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>
    );
  }
}
```

Which would result in a display like Figure 3.

Figure 3: RGBA texture, with R and G according to their respective u or v texture coordinate. From [10].

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."[30] A more detailed discussion on the targets is in section 5.10. 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 22, 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" [7] While theoretically not necessary to fulfill most requirements, as the implementation is compatible with all modern web browsers, providing 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 15)
- 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." [15]. 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:

```
1 // Card, CardContent, Tooltip and Button are all components
  ↪  provided by MaterialUI.
2 // this refers to the Zoom Plugin's controller which
3 // the content will be automatically refreshed if the refered
  ↪  values change
4 const Zoom = Component(function ZoomSlider (props) {
5     return <Card style={{ width: '100%' }} >
6         <CardContent>
7             <Tooltip title='Reset'>
8                 <Button onClick={this.resetZoom} style={{
                   ↪  marginLeft: '-8px' }}>Zoom</Button>
```

```
9               </Tooltip>
10              <Tooltip title={this.scale}>
11                  <Slider value={this.scale}
                    ↪  onChange={this.onSlider} min={0.01}
                    ↪  max={30} />
12              </Tooltip>
13              <Tooltip title='Reset'>
14                  <Button onClick={this.resetPan} style={{
                    ↪  marginLeft: '-11px' }}>Pan</Button>
15              </Tooltip>
16              <Tooltip title={this.panX}>
17                  <Slider value={this.panX}
                    ↪  onChange={this.onSliderX} min={-1 *
                    ↪  this.scale} max={1 * this.scale} />
18              </Tooltip>
19              <Tooltip title={this.panY}>
20                  <Slider value={this.panY}
                    ↪  onChange={this.onSliderY} min={-1 *
                    ↪  this.scale} max={1 * this.scale} />
21              </Tooltip>
22          </CardContent>
23      </Card>
24  })
```

It would result in a display like Figure 4.

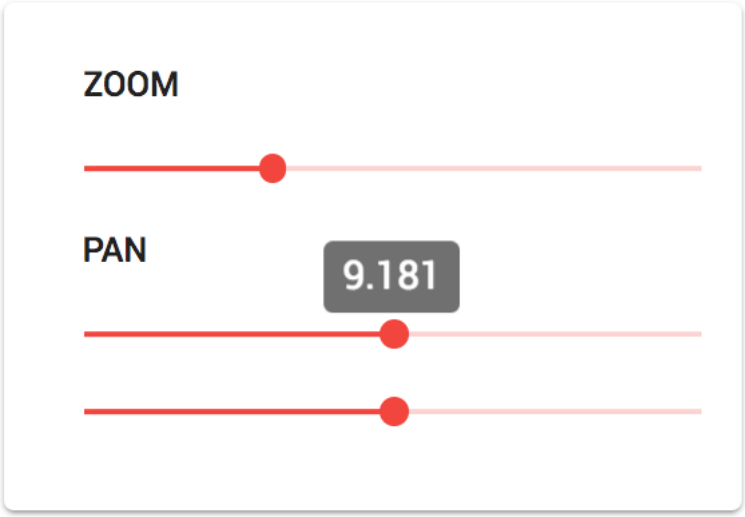


Figure 4

Misc

Further libraries of note are:

electron-webpack[8] 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[20] is providing in-browser bitwise png manipulation, required in the converter.

jszip[13] is providing in-browser zip file manipulations, which are fundamental for the BTF fileformat.

5.3 Overall Architecture

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 5.6.

- *BTFFile.tsx*, containing the fileformat implementation and utility functions, described in section 5.4.
- *Hook.tsx* and *HookManager.tsx* which provide the whole dynamic interaction system between the different plugins, shown in section 5.8.
- *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 5.9.
- *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 5.7.1.

5.4 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.

5.4.1 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.

5.4.2 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).

5.4.3 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).

5.4.4 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.

5.5 Texture Loader

5.6 State Management

Todo Text:
state management

Todo Text:
state import/export

Todo Diagramm:
redux

Todo Diagramm:
mobx actions

5.7 Renderer Stack

Todo Text:
base rendering nodes

Todo Diagramm:
stacked components

Todo Diagramm:
effects

5.7.1 Texture Loading

Todo Text:
Texture Loading

5.8 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:

```ts
// 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,
...
}
```

5.9 Plugins

Todo Text:
Plugins API

5.9.1 Base Plugin

Todo Text:
Base Plugin

5.9.2 BaseTheme Plugin

Todo Text:
Basetheme Plugin

5.9.3 RedTheme Plugin

5.9.4 TabView Plugin

```
type Tab = {
    content: PluginComponentType
    tab: TabProps,
    padding?: number,
    beforeFocusGain?: () => Promise<void>,
    afterFocusGain?: () => Promise<void>,
    beforeFocusLose?: () => Promise<void>,
    afterFocusLose?: () => Promise<void>,
}

type ActionBar = {
    onClick: () => void,
    title: string,
    enabled: () => boolean,
    tooltip?: string,
}

type ViewerTabFocus = {
    beforeGain?: () => void,
    beforeLose?: () => void,
}

type ScreenshotMeta = {
    key: string,
    fullshot?: () => (string | number)[] | string | number,
    snapshot?: () => (string | number)[] | string | number,
}

type ViewerFileAction = {
    tooltip: string,
    text: string,
    action: () => Promise<void>,
}
```

Todo Text:
TabView Plugin

5.9.5 SingleView Plugin

Todo Text:
SingleView Plugin

5.9.6 Converter Plugin

Todo Text:
Converter Plugin

5.9.7 PTMConverter Plugin

Todo Text:
PTMConverter Plugin

5.9.8 Renderer Plugin

```
type BaseNodeConfig = {
    channelModel: ChannelModel,
    node: PluginComponentType<BaseNodeProps>,
}

type RendererNode = {
    component: PluginComponentType,
    inversePoint?: (point: Point) => Point,
}

type MouseConfig = {
    listener: MouseListener,
    mouseLeft?: () => void,
}
```

Todo Text:
Renderer Plugin

Todo Text:
Base Node

Todo Text:
WebGL texture packing

5.9.9 PTM Renderer Plugin

Todo Text:
PTM Renderer Plugin

Todo Text:
Dynamic Shaders

Todo Text:
RGB vs LRGB

5.9.10 Light Control Plugin

Todo Text:
Light Control Plugin

5.9.11 Rotation Plugin

Todo Text:
Rotation Plugin

5.9.12 Zoom Plugin

Todo Text:
Zoom Plugin

5.9.13 QuickPan Plugin

Todo Text:
Zoom Plugin

5.9.14 Paint Plugin

Todo Text:
Zoom Plugin

5.9.15 Import Export Plugin

Todo Text:
Automatic Import Export

5.10 Applications

Todo Text:
Other related graphics

Todo Text:
Applications

5.10.1 Standalone Website

Todo Text:
Standalone Website

5.10.2 Embeddable

Todo Text:
Embeddable

5.10.3 Electron

Todo Text:
Electron App deliverable

6 Results

6.1 Featureset

Todo Text:
Featureset Comparison

Todo Diagramm:
Screeshots

6.2 Performance

Todo Text:
Performance

6.3 Testing

Todo Text:
Testing

Todo Text:
Shader Interpolation

Todo Text:
Image comparison

6.4 Rollouts and Deployments

Todo Text:
Rollout

Todo Text:
Non-Tech deployment

o

7 Discussion

7.1 Community Onboarding

Todo Text:
Community Onboarding

7.2 Novelties

Todo Text:
Novelties results

7.3 Future Work

The future work can be split into two parts. Improvements of the current system, including better performance and bug fixes, and further extensions with new functionality.

7.3.1 WebGL 2

Todo Text:
Future Work

8 Conclusion

Todo Text:
Conclusion

References

[1] *Alternative syntax madness · Issue #487 · mobxjs/mobx-state-tree*. URL: https://github.com/mobxjs/mobx-state-tree/issues/487.

[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] Library of Congress. *Polynomial Texture Map (PTM) File Format*. June 14, 2018. URL: https://www.loc.gov/preservation/digital/formats//fdd/fdd000487.shtml (visited on 08/10/2018).

[5] Library of Congress. *Reflectance Transformation Imaging (RTI) File Format*. June 9, 2018. URL: https://www.loc.gov/preservation/digital/formats//fdd/fdd000486.shtml#notes (visited on 08/10/2018).

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

[7] *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).

[8] *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).

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

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

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

[12] 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).

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

[14] Tom Malzbender and Dan Gelb. "Polynomial Texture Map (.ptm) File Format". In: (), p. 6.

[15] *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).

[16] *mobx: Simple, scalable state management*. Aug. 13, 2018. URL: https://github.com/mobxjs/mobx (visited on 08/13/2018).
[17] *mobx-state-tree: Model Driven State Management*. Aug. 20, 2018. URL: https://github.com/mobxjs/mobx-state-tree (visited on 08/21/2018).
[18] *React - A JavaScript library for building user interfaces*. URL: https://reactjs.org/index.html (visited on 08/13/2018).
[19] 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).
[20] Arian Stolwijk. *pngjs: Pure JavaScript PNG decoder*. July 20, 2018. URL: https://github.com/arian/pngjs (visited on 08/21/2018).
[21] *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).
[22] *TSLint*. URL: https://palantir.github.io/tslint/ (visited on 08/17/2018).
[23] *TSLint - Visual Studio Marketplace*. URL: https://marketplace.visualstudio.com/items?itemName=eg2.tslint (visited on 08/17/2018).
[24] *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).
[25] *Visual Studio Code - Code Editing. Redefined*. URL: http://code.visualstudio.com/ (visited on 08/17/2018).
[26] *WebGL Specification*. URL: https://www.khronos.org/registry/webgl/specs/1.0/ (visited on 08/21/2018).
[27] *WebGL Stats*. URL: http://webglstats.com/ (visited on 08/21/2018).
[28] *WebGL Stats Texture Units*. URL: http://webglstats.com/webgl/parameter/MAX_TEXTURE_IMAGE_UNITS (visited on 08/21/2018).
[29] *WebGL tutorial*. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial (visited on 08/21/2018).
[30] *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).