Getting Started With Vue.js

***5th December, 2016**: This tutorial covers Vue.js 1.x. If you're looking for Vue 2 content, see here: Getting up and Running with the Vue.js 2.0 Framework.*

***21st July, 2016**: Article has been updated to cover Vue.js 1.0.x, and a section on components added.*

Vue.js is a JavaScript library that helps you build web applications using the the MVVM (Model-View-ViewModel) architectural pattern. At first glance, it might seem quite similar to AngularJS, but once you start working with it you'll quickly realize that Vue.js is not only much simpler and easier to learn, but also more flexible.

In this introductory tutorial, I'll teach you the basic concepts of Vue.js, and give a complete overview of its most important features.

> Vue.js 1.0.x has a few syntax changes that are not compatible with Vue.js 0.12.x. If you have experience using those early versions, you might have already noticed some of the changes in this tutorial. You can get an overview of all the changes here: What's New in Vue.js 1.0

Adding Vue.js to Your Page

Though you can get the latest release of Vue.js from GitHub, you might find it easier to load it from a CDN:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.26/vue.min.js">
</script>
```

Creating a View-Model

In Vue.js, view-models are created using the `Vue` class. If you are wondering what a view-model is, you can think of it as an object that makes it very easy for you to display your model's data inside a view (you can treat any object literal as a model, and any HTML UI element as a view).

To see how a view-model works, let's start by creating a view. An empty `<div>` will do for now:

```
<div id="my_view"> </div>
```

And here's an object literal which will be our model. As it deals with JavaScript code, make sure you create it inside a `<script>` tag, or in a separate JS file.

```
var myModel = { name: "Ashley", age: 24 };
```

Now that we have a view and a model, it's time to create a view-model (a `Vue` instance) that binds both together:

```
var myViewModel = new Vue({ el: '#my_view', data: myModel });
```

As you can see, the `el` property points to the view (you can use any CSS selector here), and the `data` property points to the model. Our view-model is now ready to be used.

To display your model's data inside the view, you should use mustache-style bindings. For example, to display the `age` property of your model you would add the string `{{ age }}` inside your view. The following code snippet uses both the properties of the model:

```
<div id="my_view">{{ name }} is {{ age }} years old.
</div> <!-- Evaluated to "Ashley is 24 years old" -->
```

Any changes you make to your model will be instantly visible in the view.

Creating Two-Way Bindings

The mustache-style binding we used in the previous example is a one-way binding. This means that it can only show the data of the model, not modify it. If you want to allow the view to edit the model you should create a two-way binding instead, using the `v-model` directive.

Let's change our view so that it contains an `input` element whose `v-model` points to the `name` property:

```
<div id="my_view"> <label for="name">Enter name:</label> <input type="text" v-
model="name" id="name" name="name" /> <p>{{ name }} is {{ age }} years old.</p>
</div>
```

At this point, if you edit the value in the input field, your model will change immediately.

Using Filters

A filter is a function you can use inside a directive or a mustache-style binding. A filter is always preceded by a pipe symbol (`|`). For example, if you want to display the `name` property in uppercase, your mustache-style binding would look like this:

```
{{ name | uppercase }}
```

Also in this case, there's a demo for filters that you can examine:

The filters `lowercase` and `capitalize` can be used in a similar manner.

We'll talk more about filters in the next section.

Rendering Arrays

If your model has an array, you can display the contents of that array by adding a `v-for` directive to a list's `<li>` element. To demonstrate that, let's add an array to our model:

```
var myModel = { name: "Ashley", age: 24, friends: [ {name: "Bob", age: 21 }, {
name: "Jane", age: 20 }, { name: "Anna", age: 29 } ] };
```

The following code shows you how to display the `name` property of every object in the `friends` array:

```
<div id="my_view"> <ul> <li v-for="friend in friends"> {{ friend.name }} </li>
</ul> </div>
```

To change the order in which the elements are listed, use the `orderBy` filter inside the `v-for` directive. For example, if you want to order the elements by `age`, your code should be as follows:

```
<div id="my_view"> <ul> <li v-for="friend in friends | orderBy 'age'"> {{
friend.name }} </li> </ul> </div>
```

You can also render items conditionally. To achieve this task, use the `filterBy` filter. For example, the following example shows how to render only those items which contain the character "a" in the `name` field:

```
<div id="my_view"> <ul> <li v-for="friend in friends | filterBy 'a' in 'name'">
{{ friend.name }} </li> </ul> </div>
```

This third demo uses a two-way data binding and a `filterBy` filter to simulate a search:

Handling Events

In Vue.js, if you need to handle events associated with your view, you should add event handlers inside the view-model's `methods` property. Inside all Vue.js event handlers, you can use `this` to access the items in the data model.

The following view-model contains a click handler:

```
var myViewModel = new Vue({ el: '#my_view', data: myModel,
methods: { myClickHandler: function(e) { alert("Hello " + this.name); } } });
```

To associate an event handler defined in the view-model with one or more UI elements in the view, you should use the `v-on` directive. For example, the following view has a `<button>` which uses the `v-on` directive to call `myClickHandler`:

```
<div id="my_view"> Name: <input type="text" v-model="name"> <button v-
on:click="myClickHandler">Say Hello</button> </div>
```

Putting these snippets together, results in this demo for click handler:

Creating Components

Vue.js allows you to create custom HTML elements that can be used within your views. By using such elements, you can make your code not only more concise, but also more readable.

To define and register a custom HTML element, you must create a Vue component

using the `component` method of the `Vue` class. You can specify the contents of the custom element using the component's `template` property.

Here's a code snippet that defines and registers a simple custom HTML element called `<sitepoint>`.

```
Vue.component('sitepoint', { template: '<a href="https://www.sitepoint.com">Sitepoint</span>' });
```

The `<sitepoint>` element can now be used inside your view just like any other standard HTML element.

```
<div id="my_view"> <sitepoint></sitepoint> </div>
```

Standard HTML elements usually have attributes associated with them that allow you to control how they look and behave. The custom elements you create using Vue.js too can have such attributes. To specify what attributes your element can have, you must use the `props` property while creating the associated component.

Here's how you would add a prop called `channel` to the `<sitepoint>` element:

```
Vue.component('sitepoint', { props: ['channel'], template: '<a href="https://www.sitepoint.com/{{ channel | lowercase }}">{{ channel }} @Sitepoint</span>' });
```

As you can see in the code above, moustache-style bindings can be used to embed the value of a prop inside the `template`.

You are now free to use the `channel` attribute inside the `<sitepoint>` tag. For example, here's how you can use it to link to two different channels of Sitepoint:

```
<div id="my_view"> <sitepoint channel="JavaScript"></sitepoint> <sitepoint channel="Web"></sitepoint> </div>
```

Feel free to modify the code in the following demo to experiment with other templates and props.

Conclusion

In this introductory tutorial about Vue.js we've looked at how to use one-way and two-way data binding, directives, filters, and events. We've also learned how to create your own HTML elements using Vue.js components.

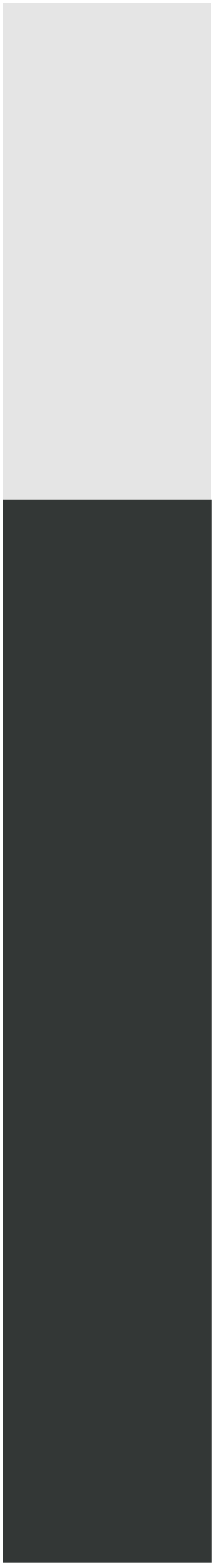
The topics covered, should be enough to start creating interactive web interfaces using this simple framework. If you are looking for more features, such as support for Ajax or routing operations, there are a growing number of Vue.js plugins available to add to your projects.

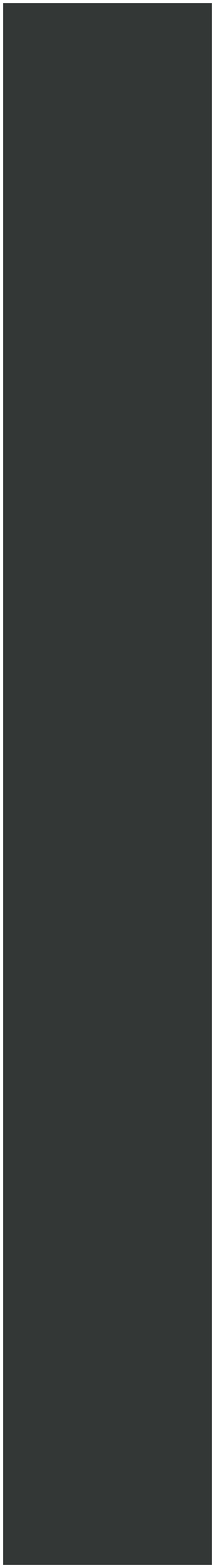
To learn more about Vue.js, I suggest going through the Vue.js API reference and also having a look at the guide available on the website.

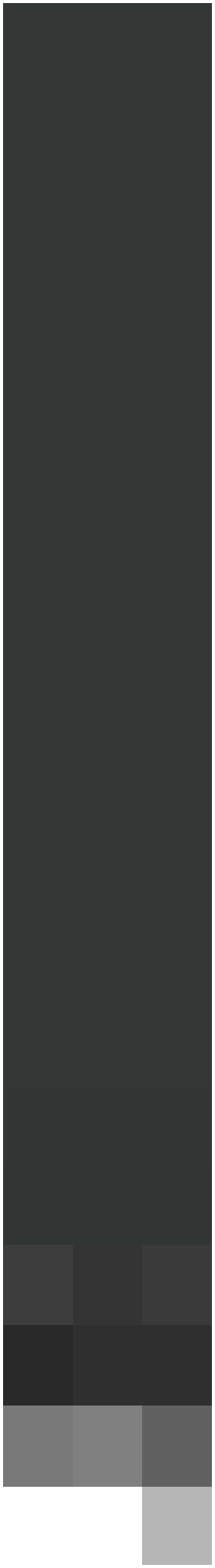
Meet the author

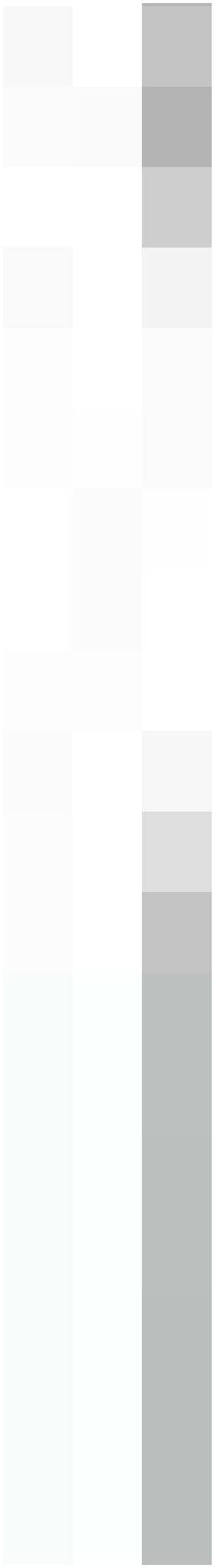
Ashraff Hathibelagal

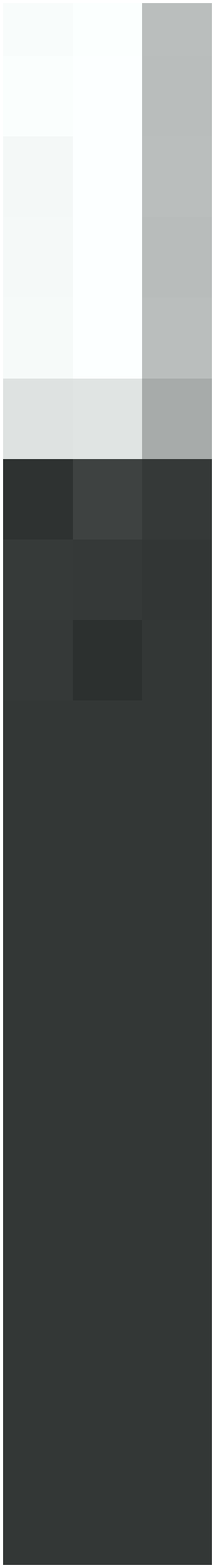
Hathibelagal is an independent developer and blogger who loves tinkering with new frameworks, SDKs, and devices. Read his blog here.

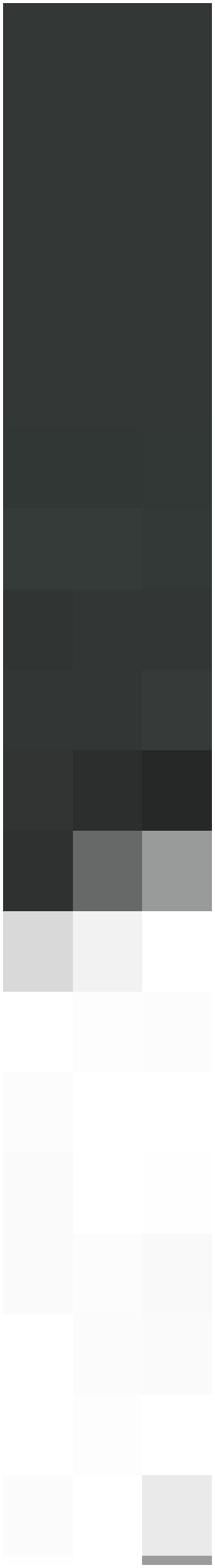


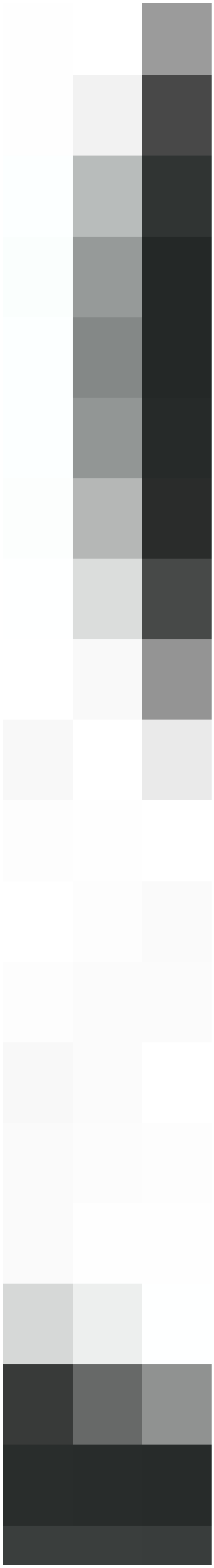


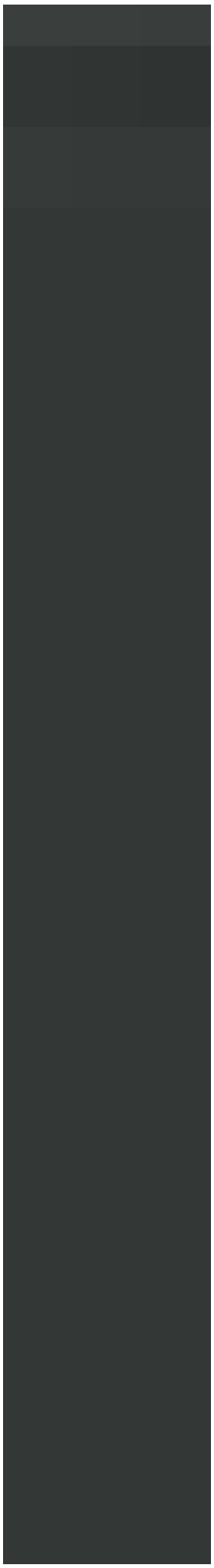


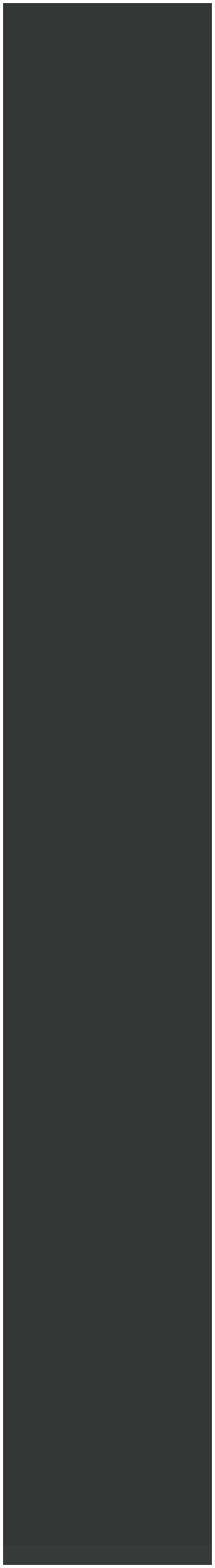


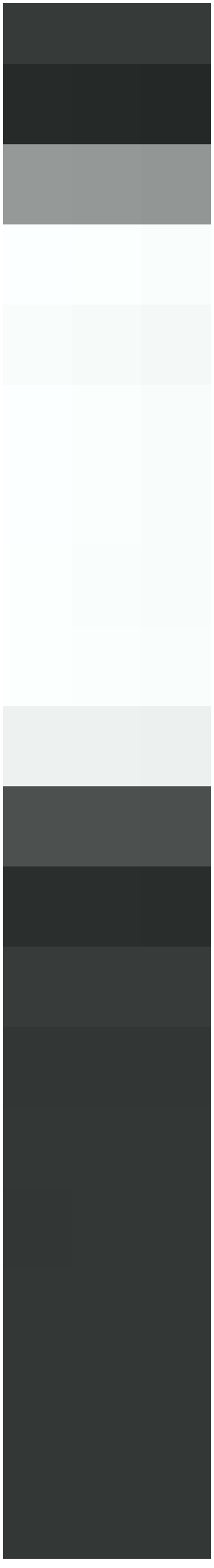


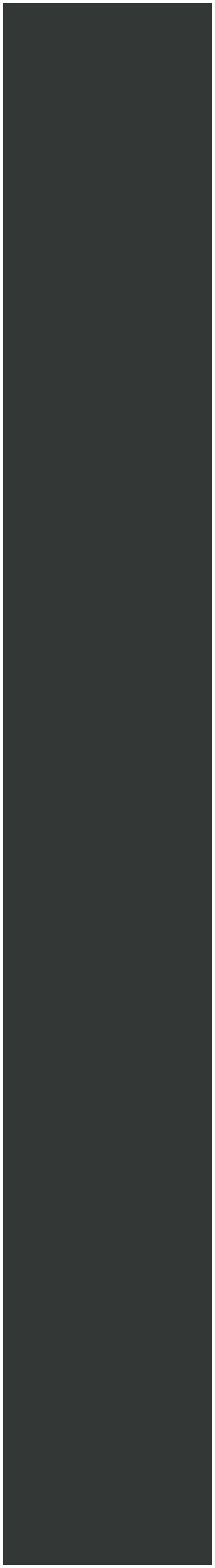


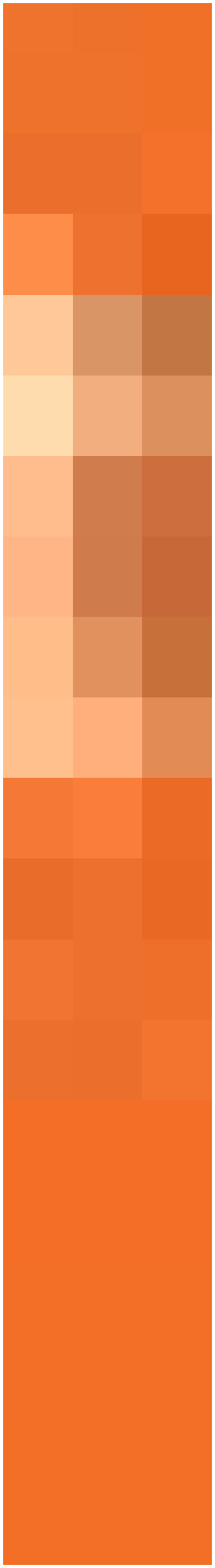




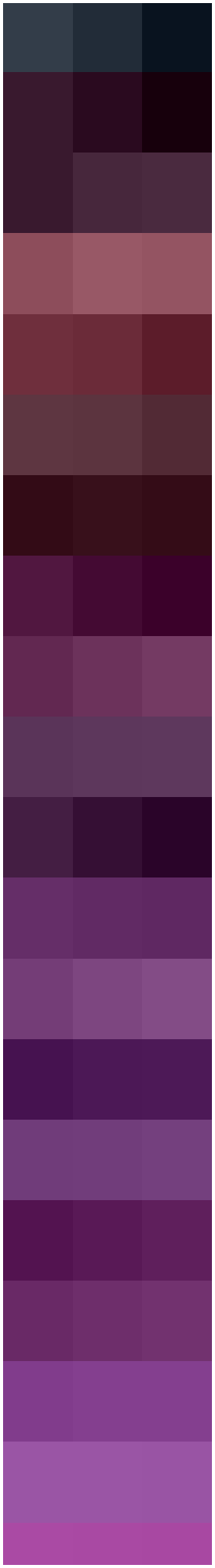


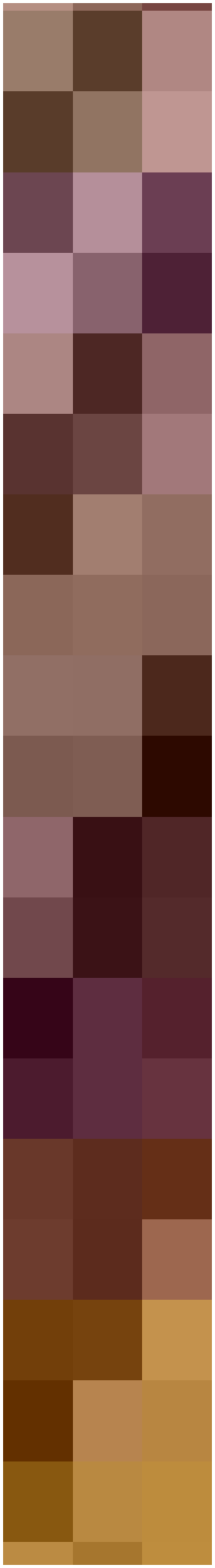


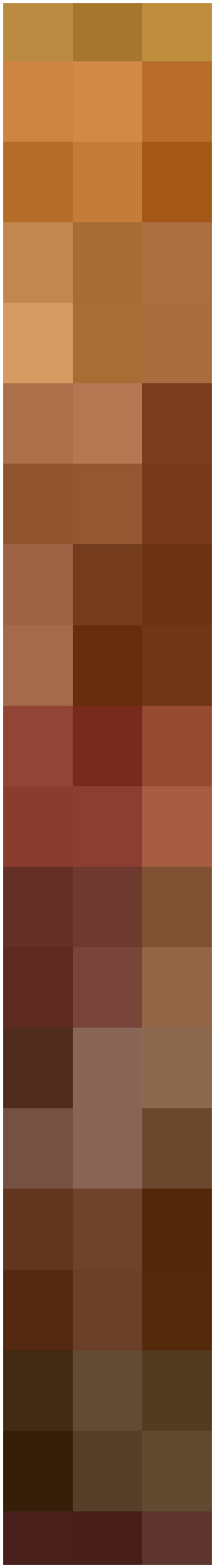


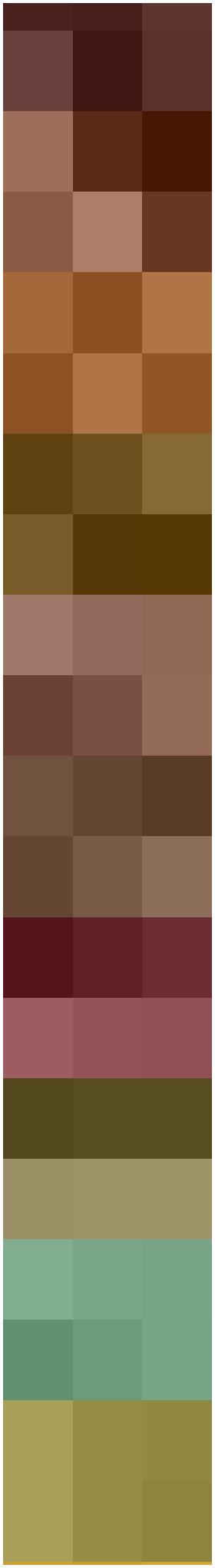


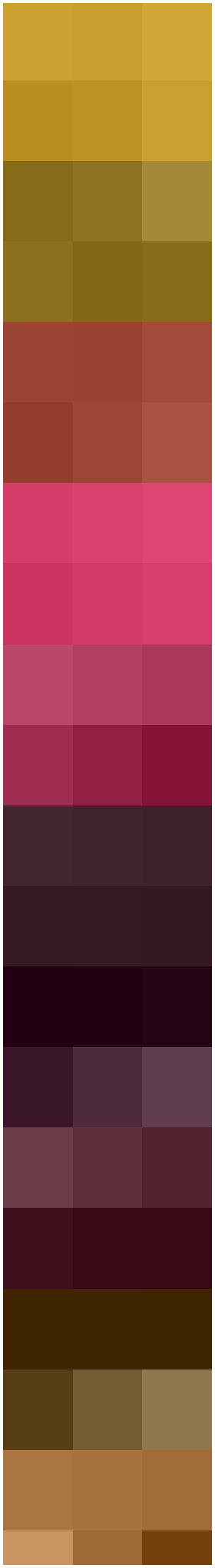
Real-Time 3D Graphics with WebGL 2 - Second Edition

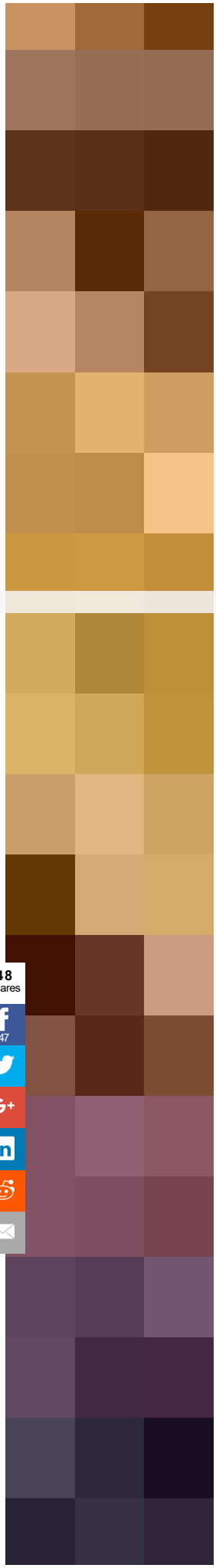


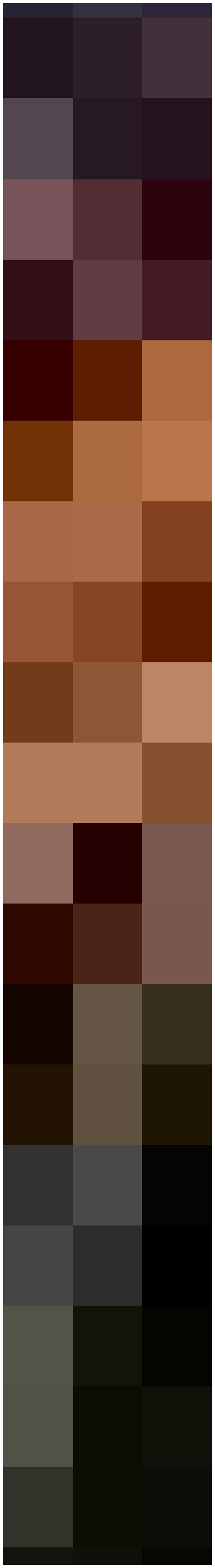


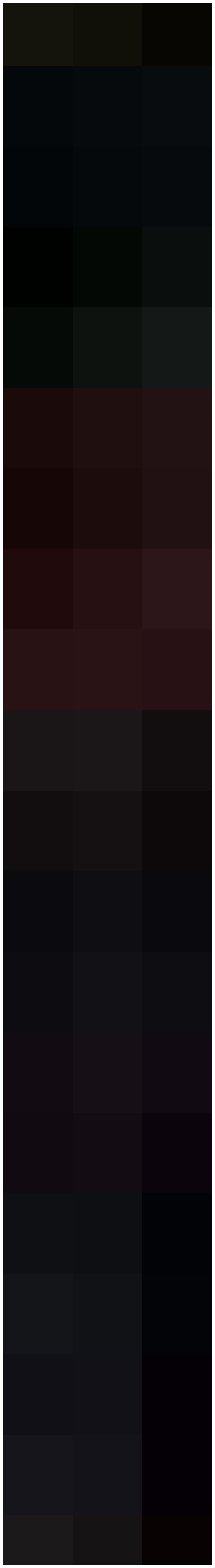


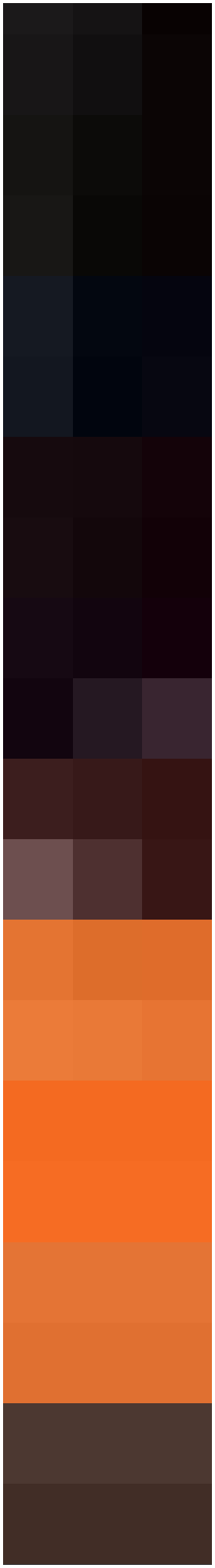


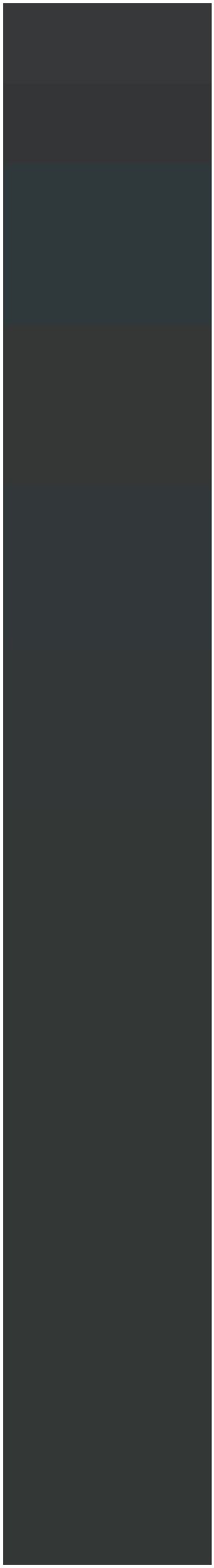


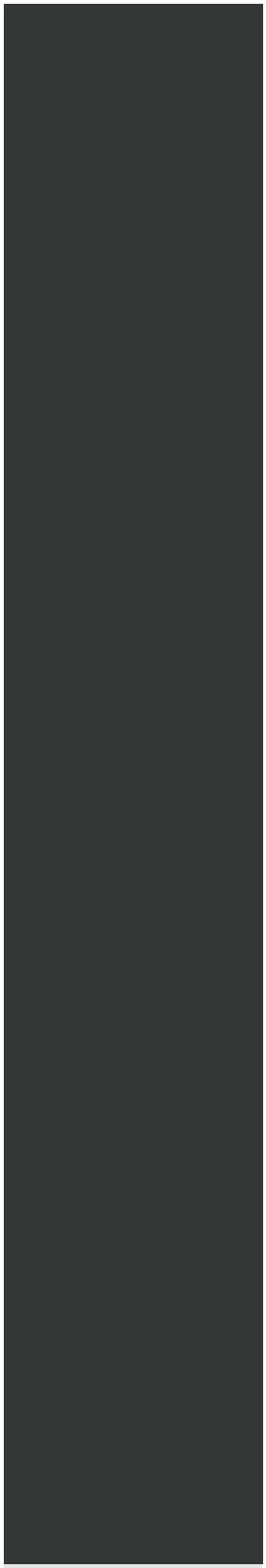




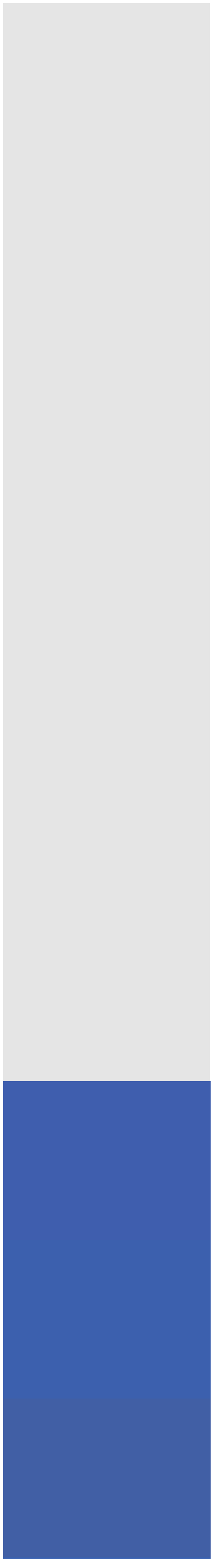


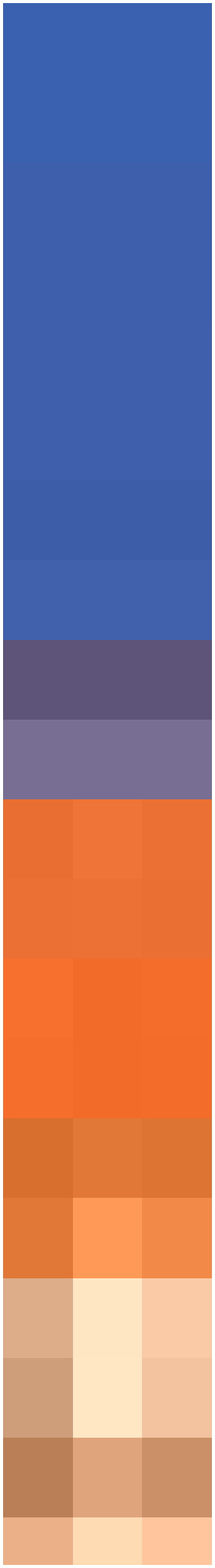


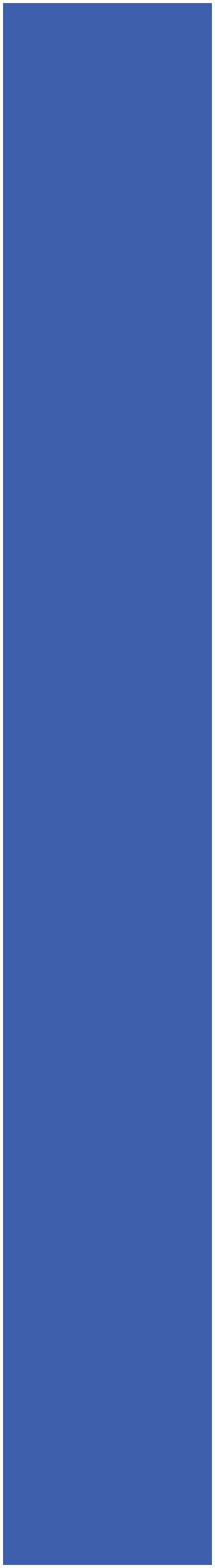


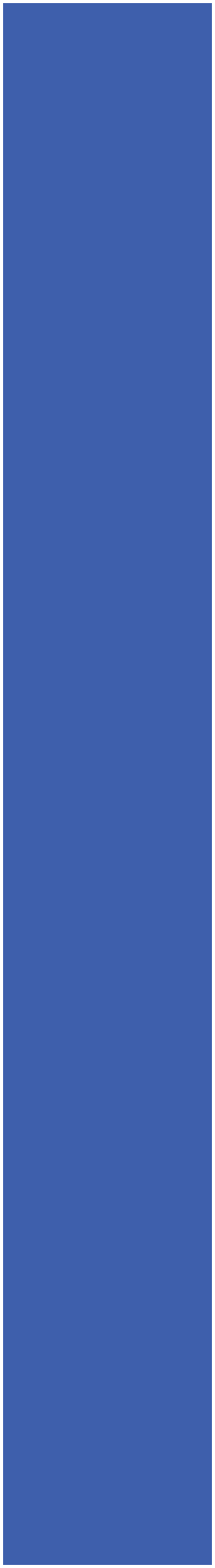


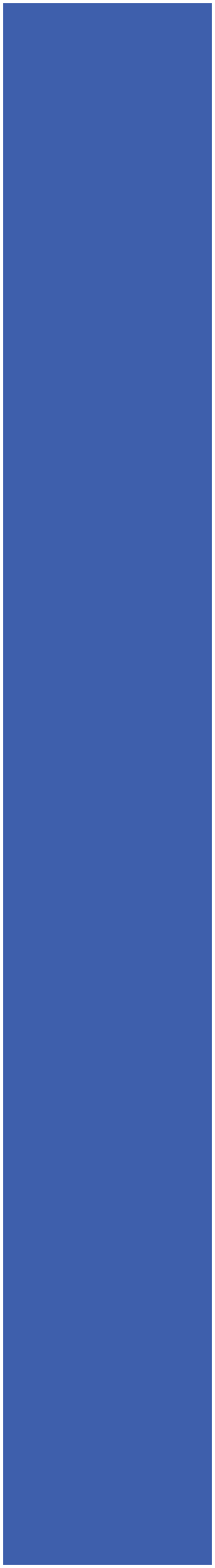
PREMIUM BOOK

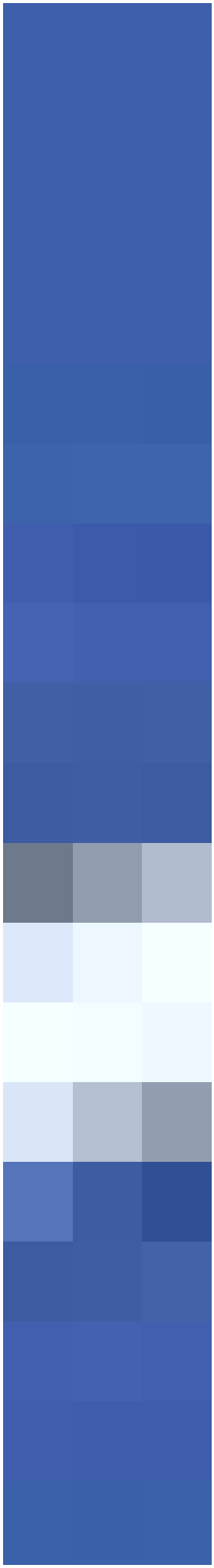


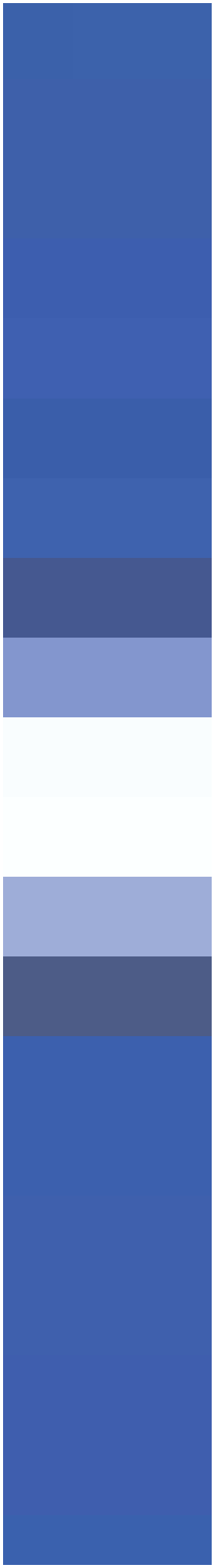


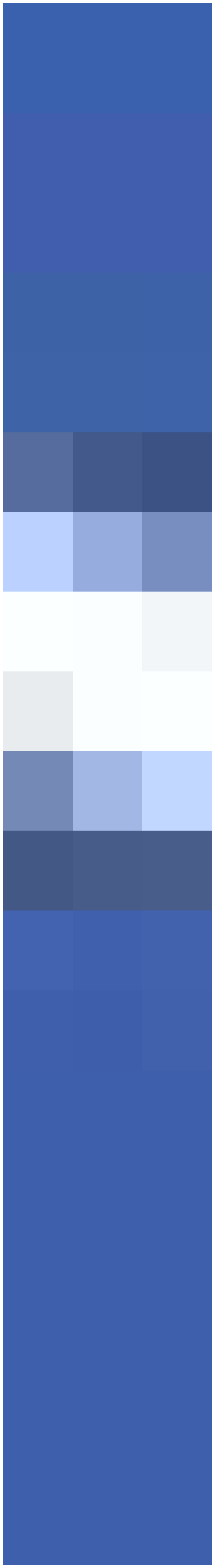


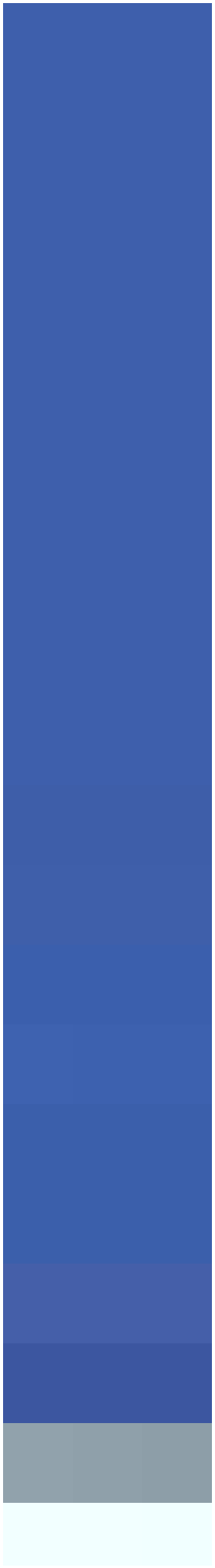




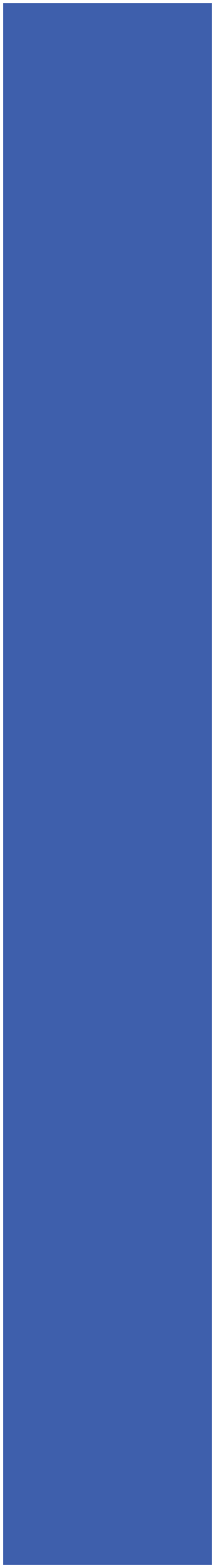


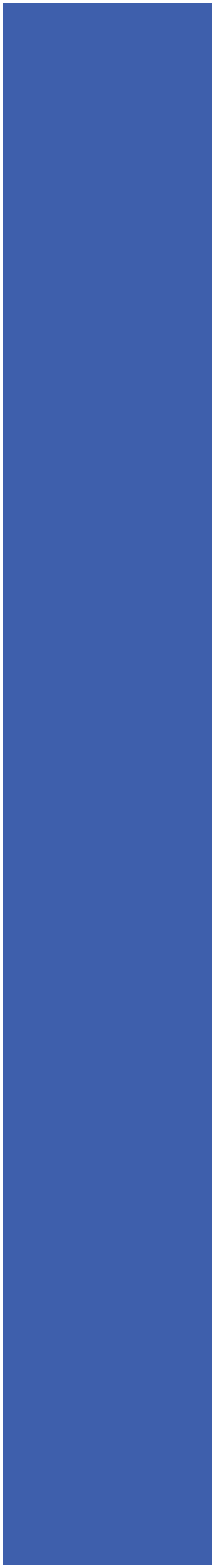


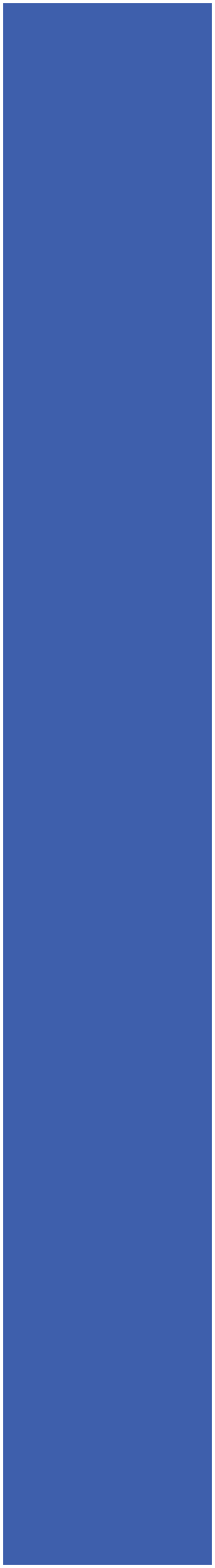


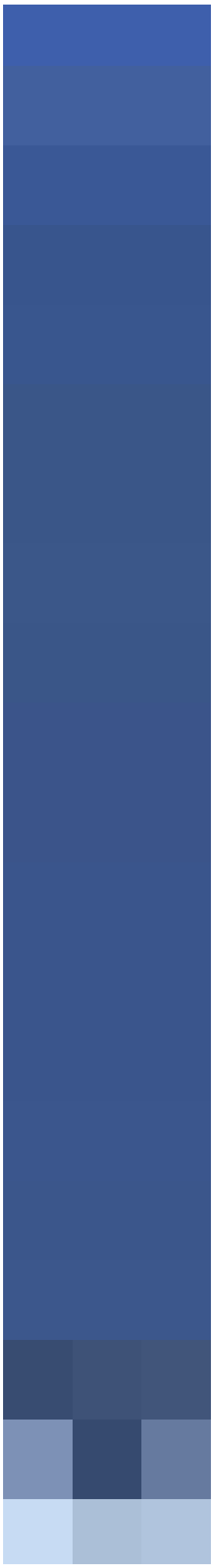


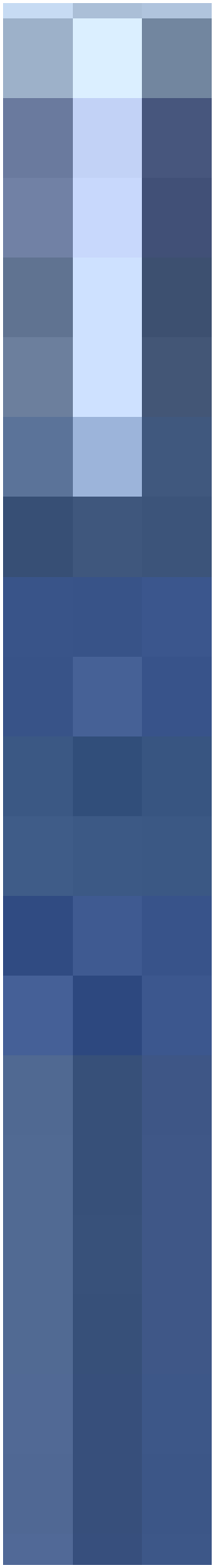
Mastering React Native

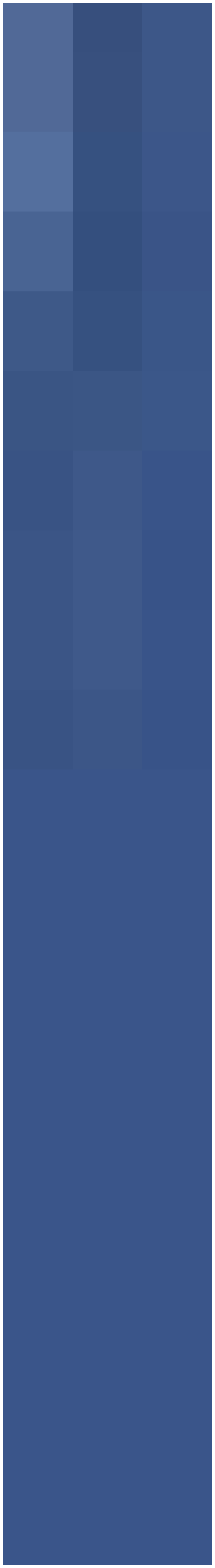


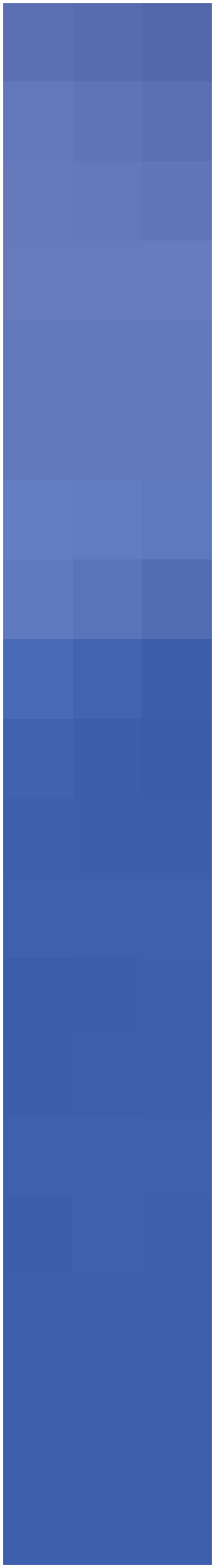


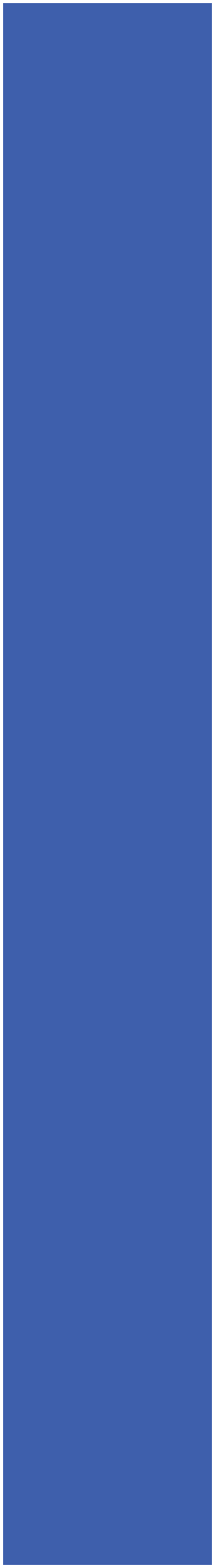


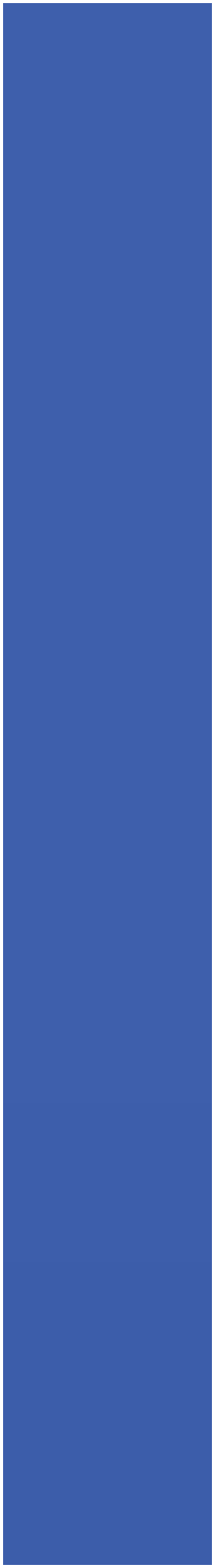


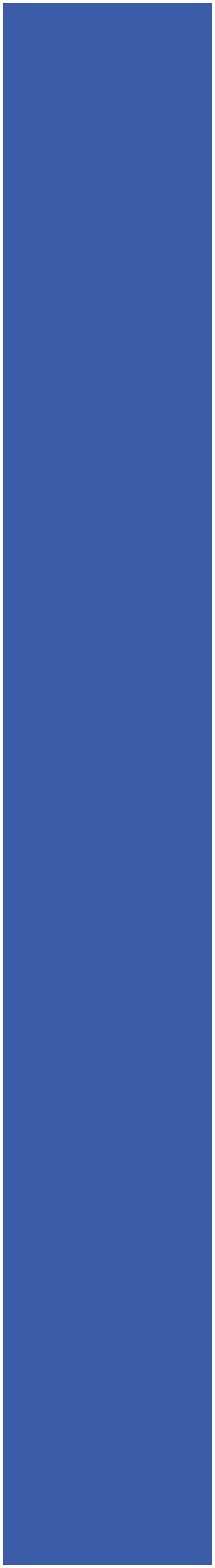




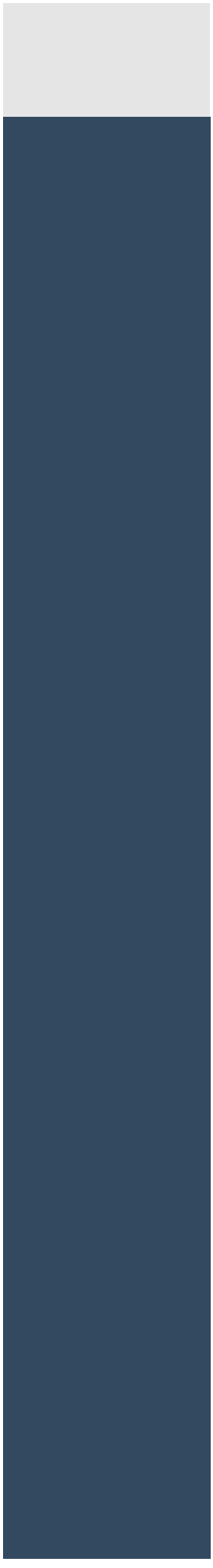


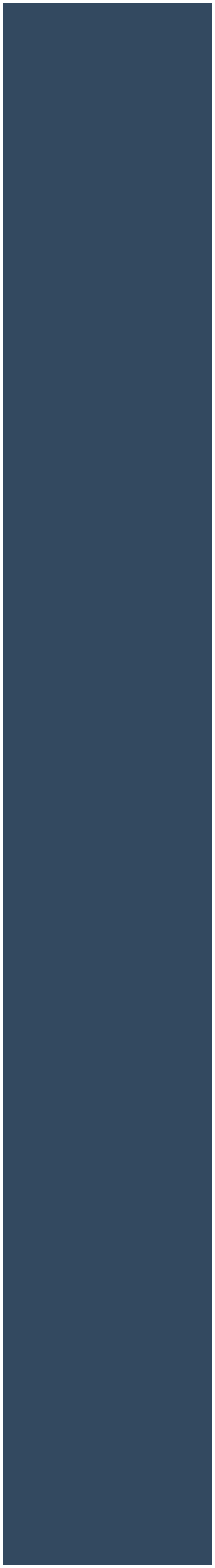


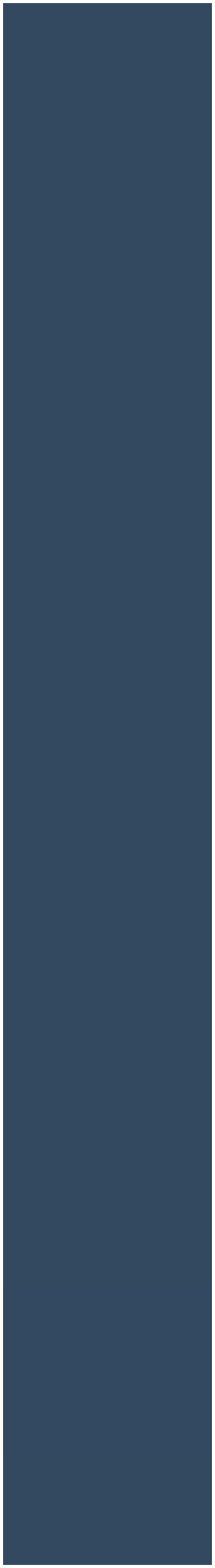


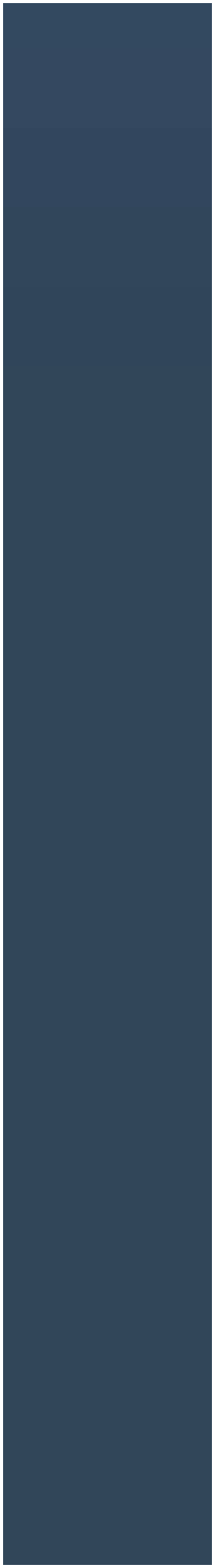


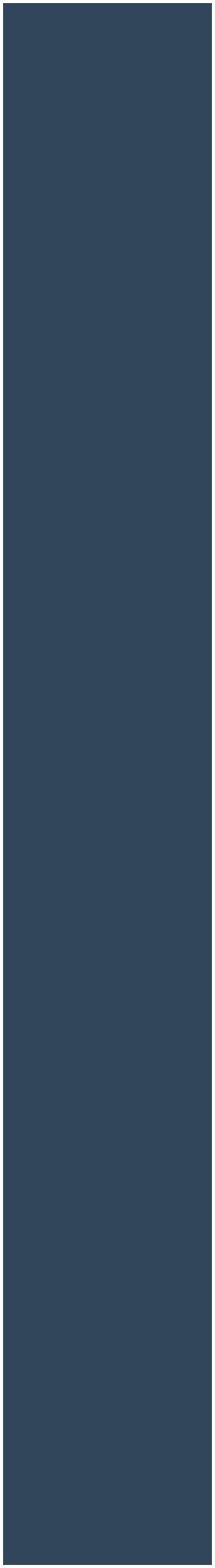
PREMIUM BOOK

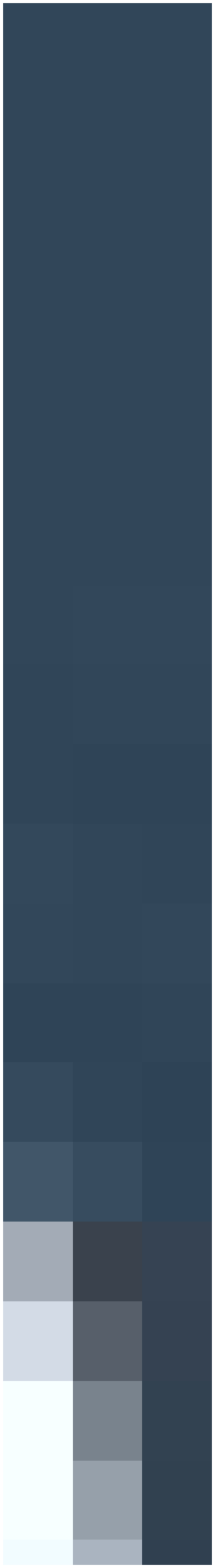


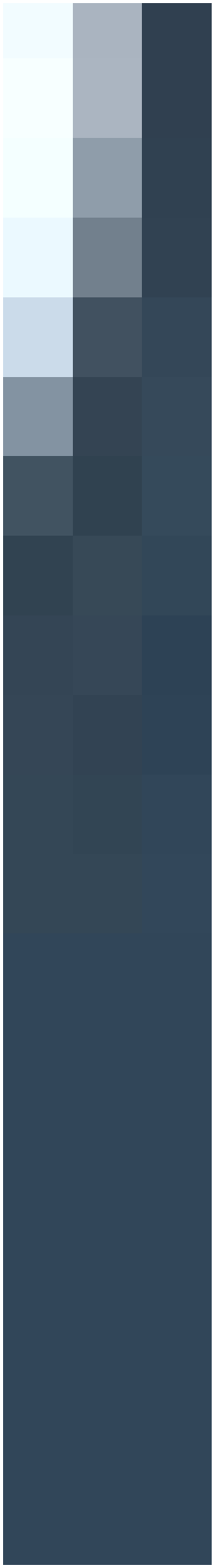


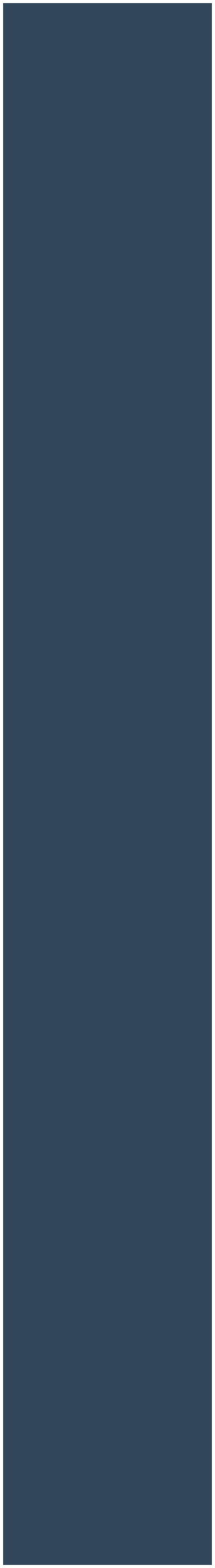


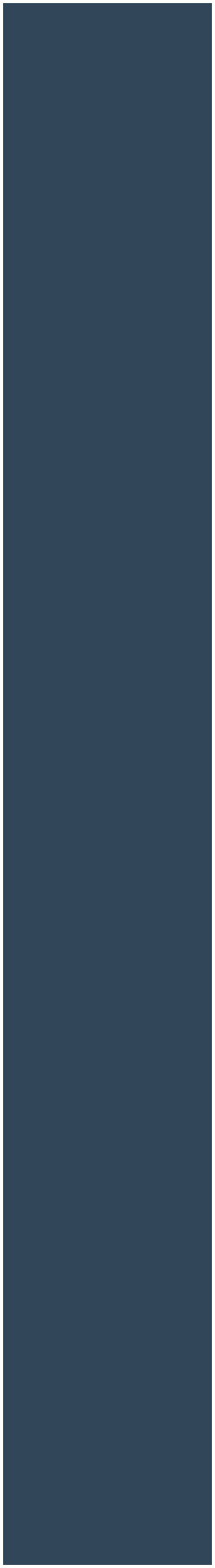


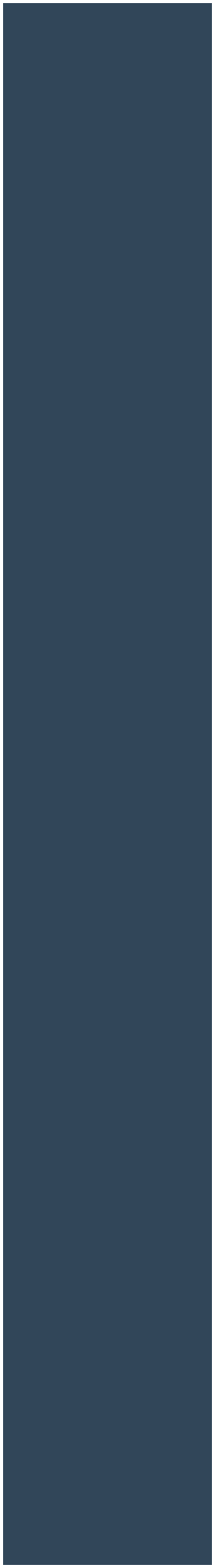


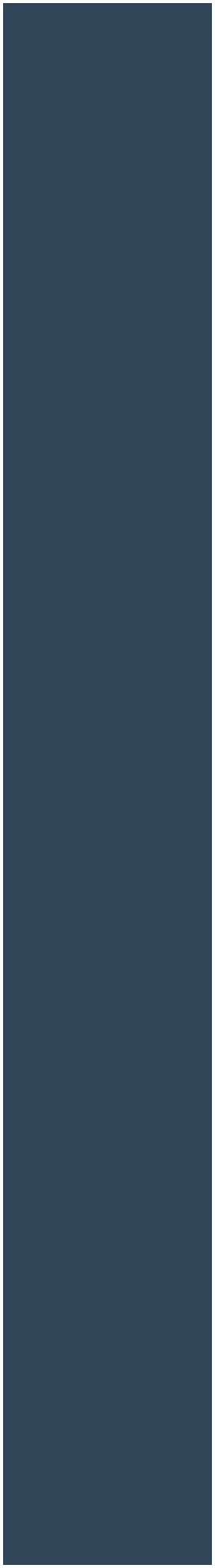


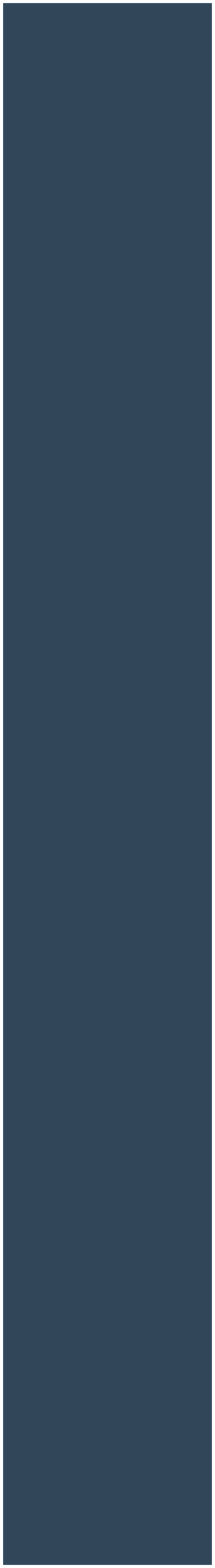




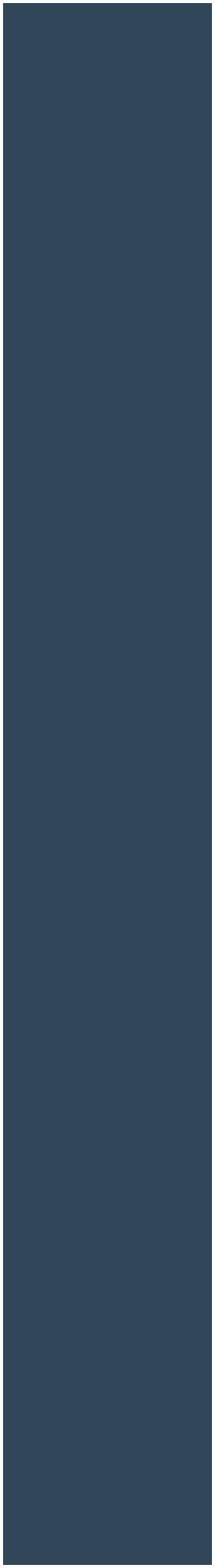


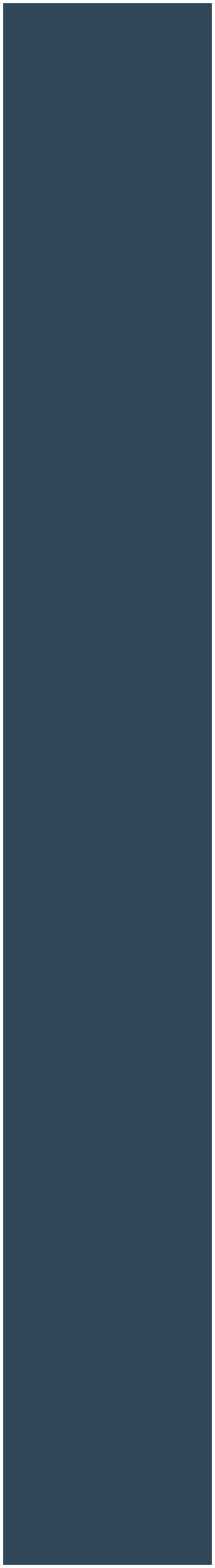


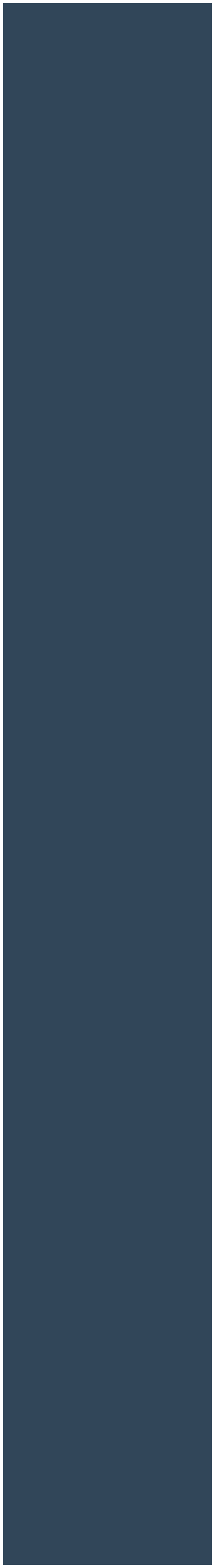


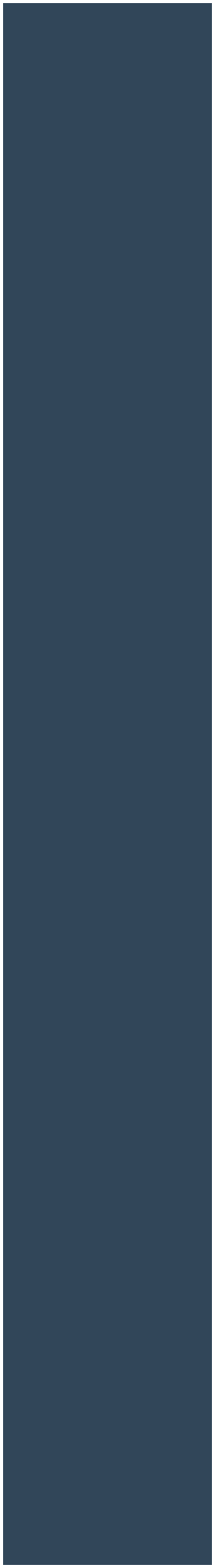


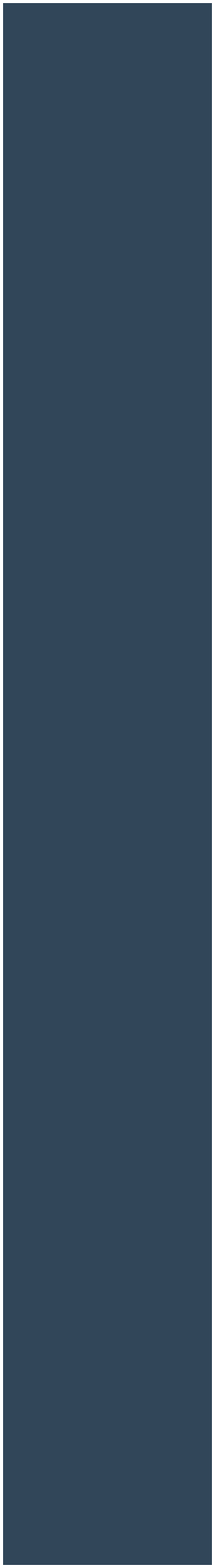
Vue.js: Tools & Skills

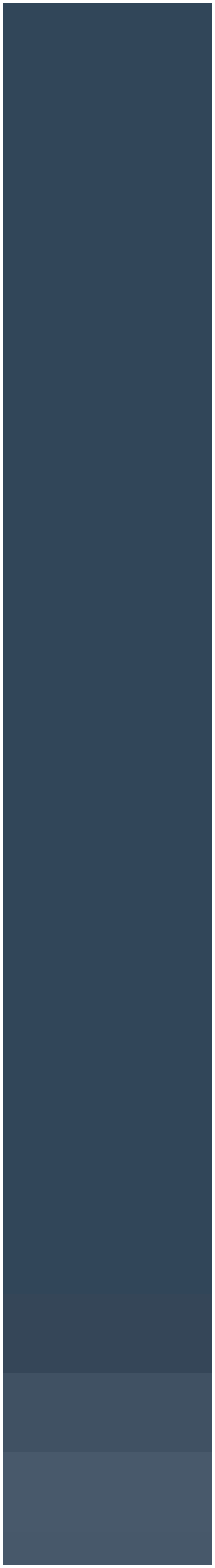


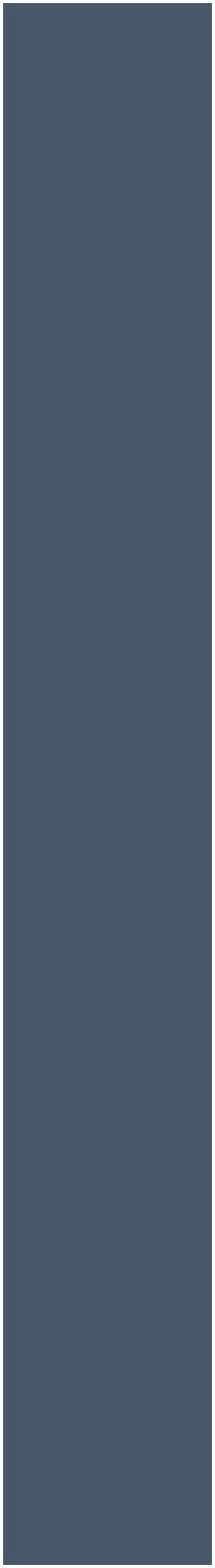


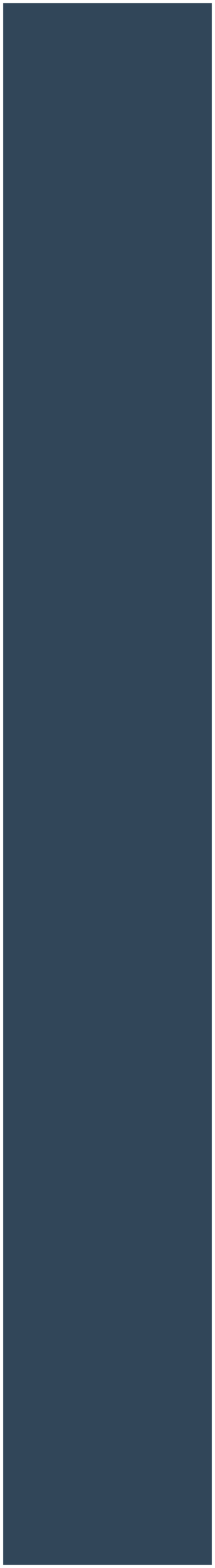


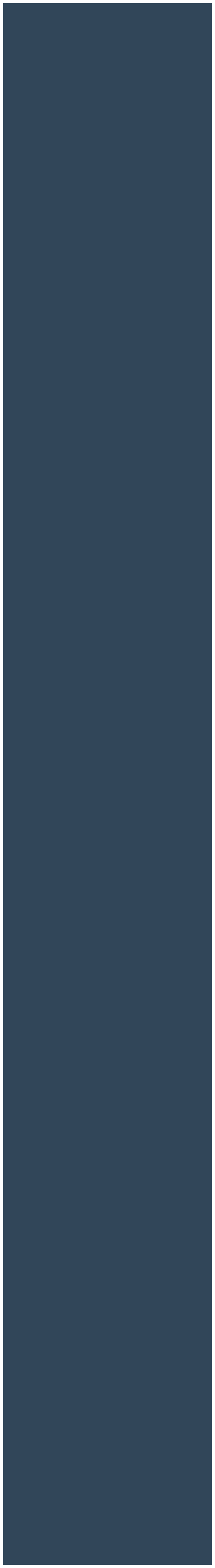


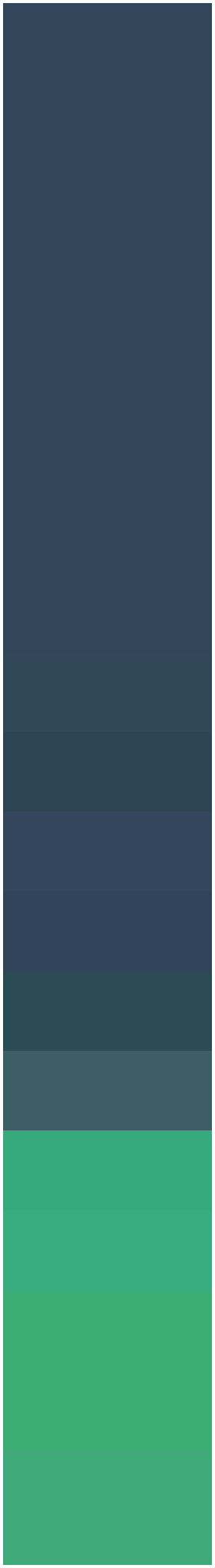


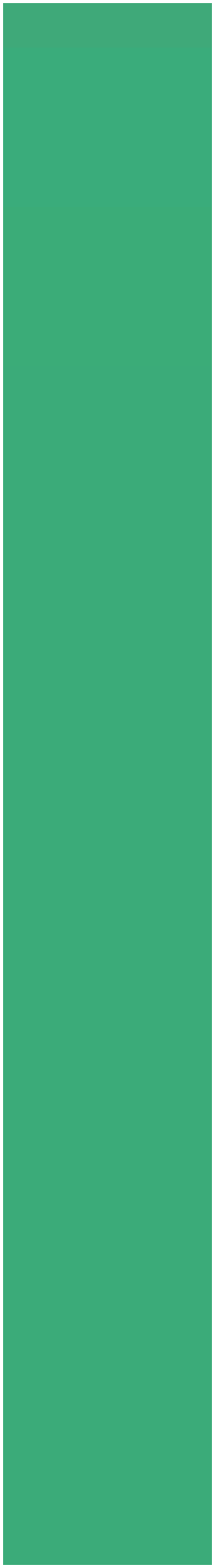


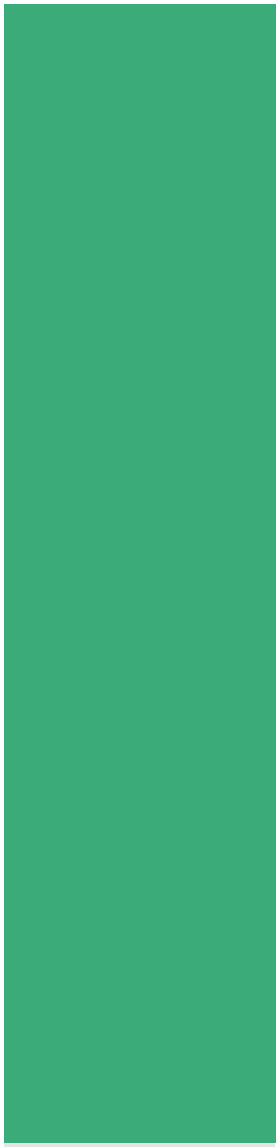


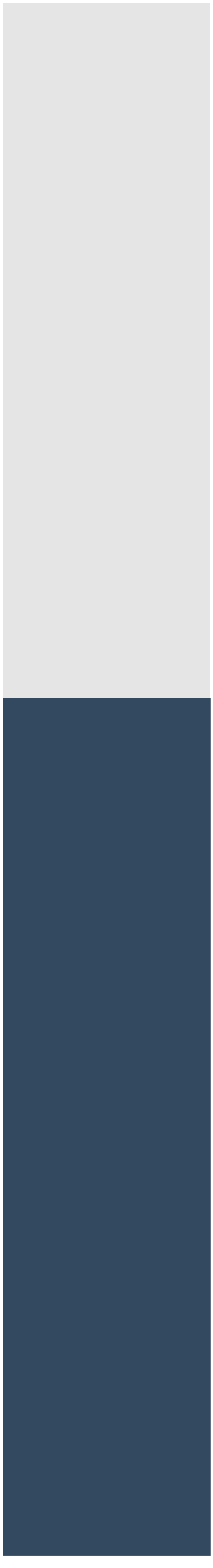


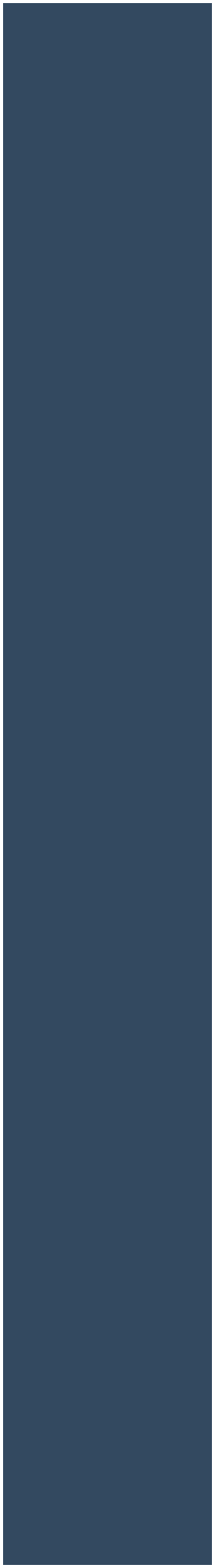


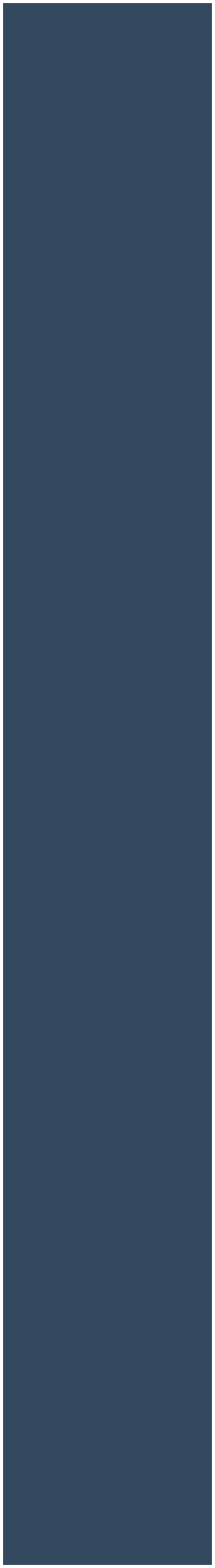


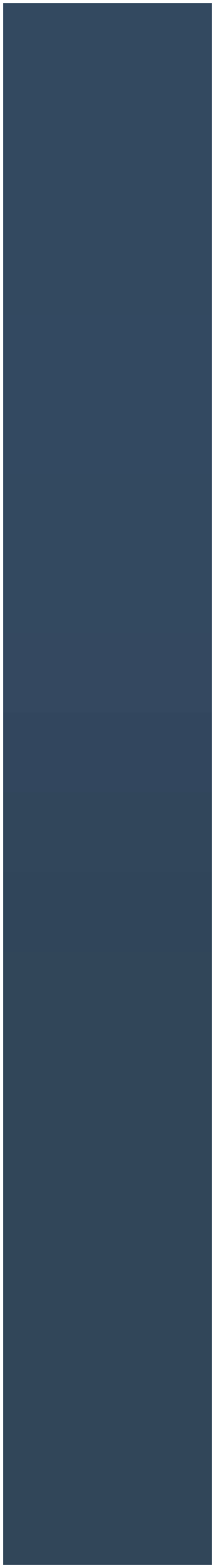


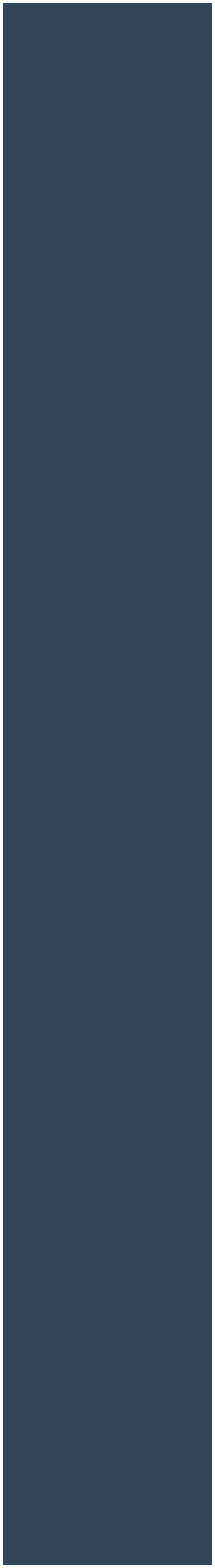


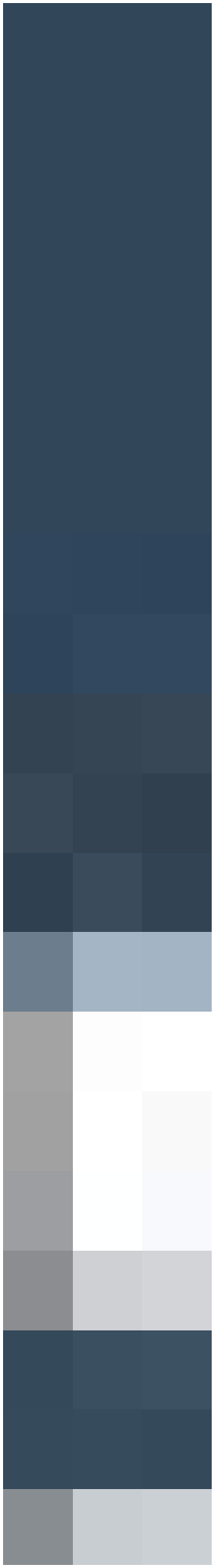


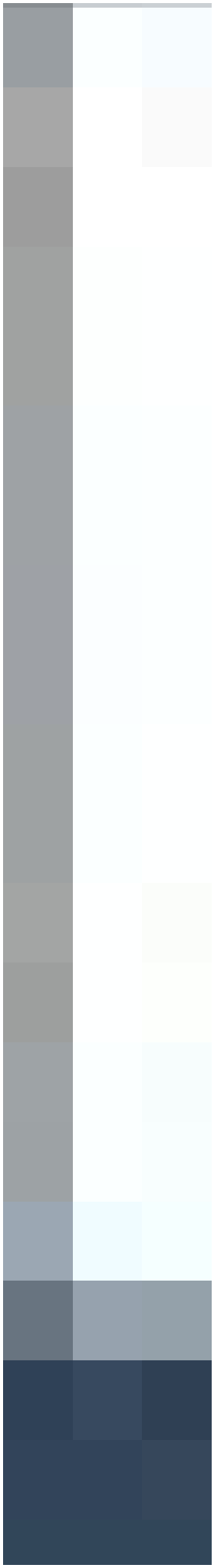


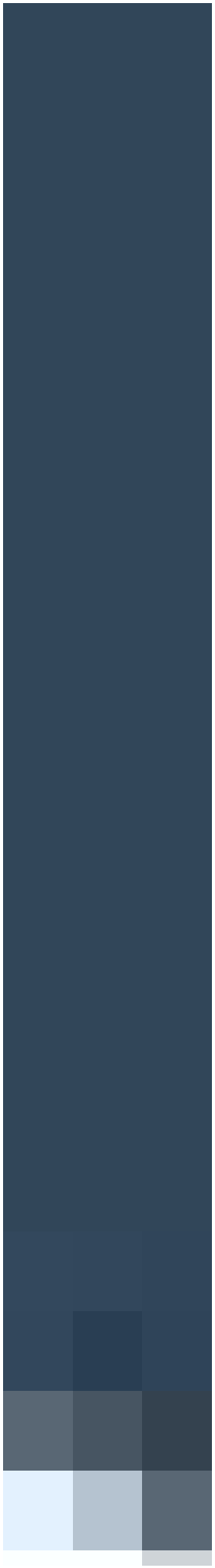


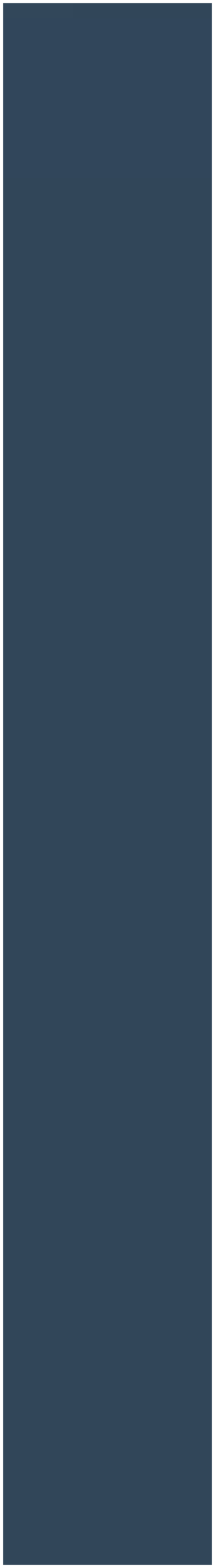


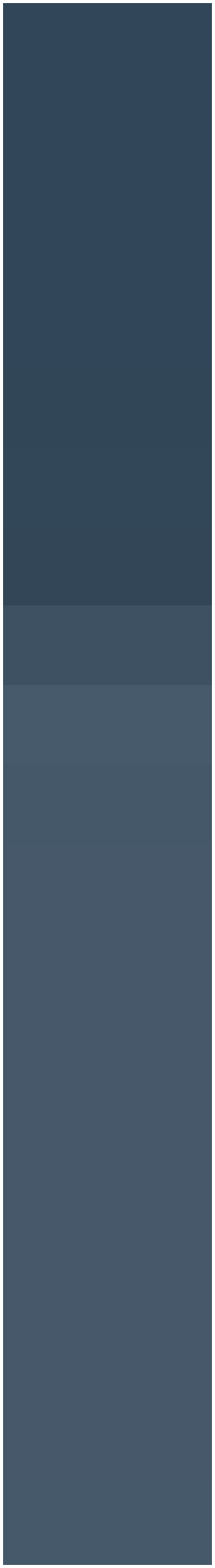


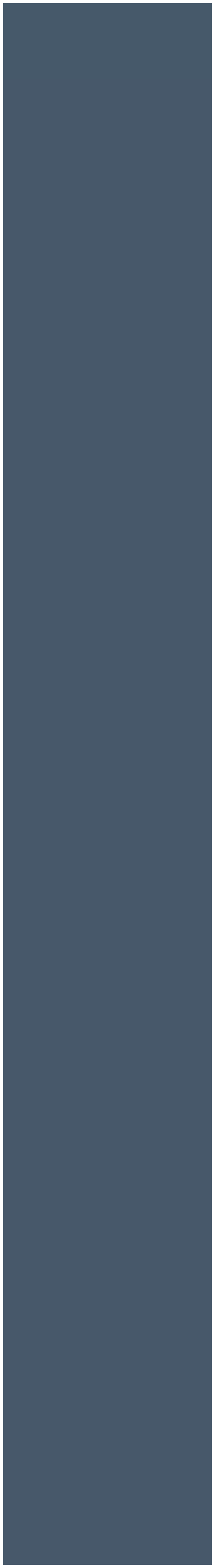


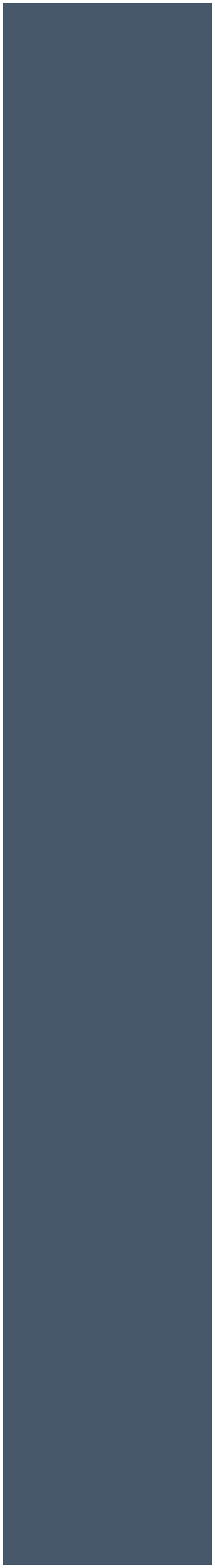




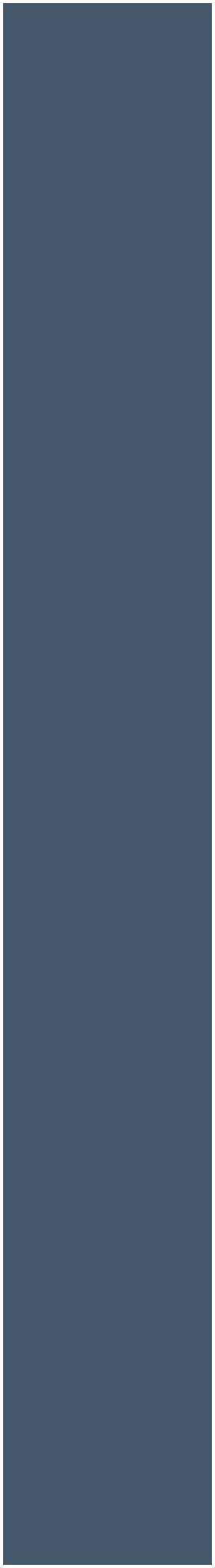


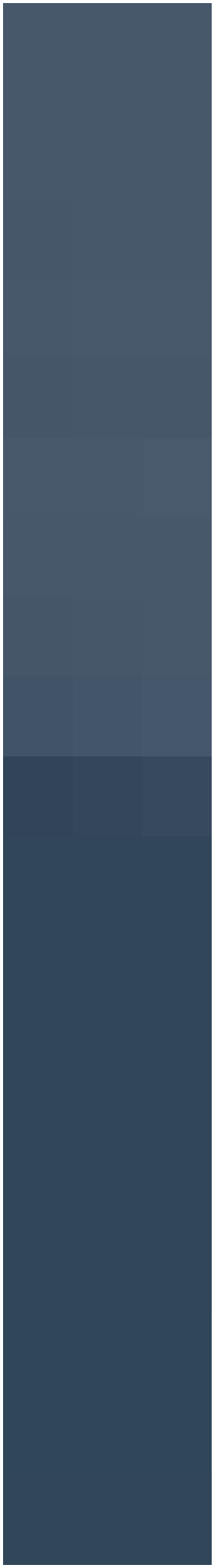


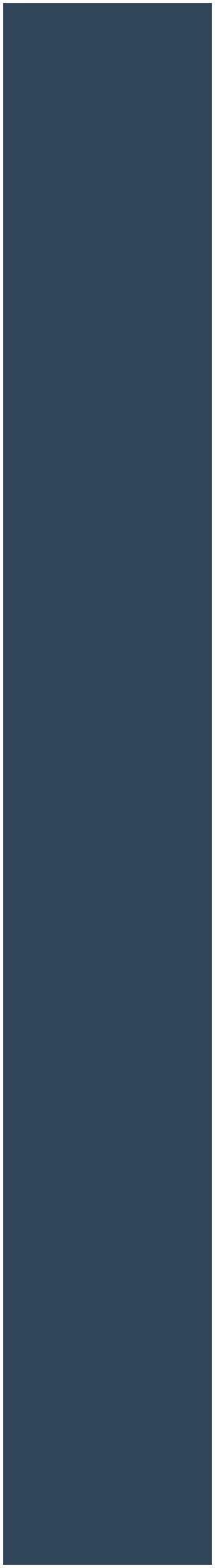


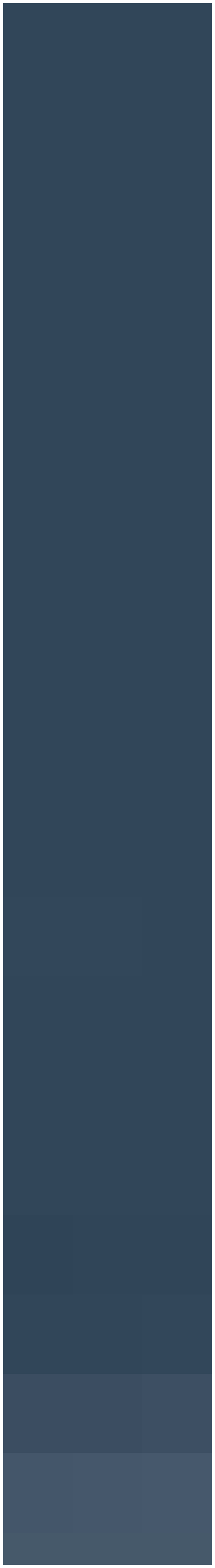


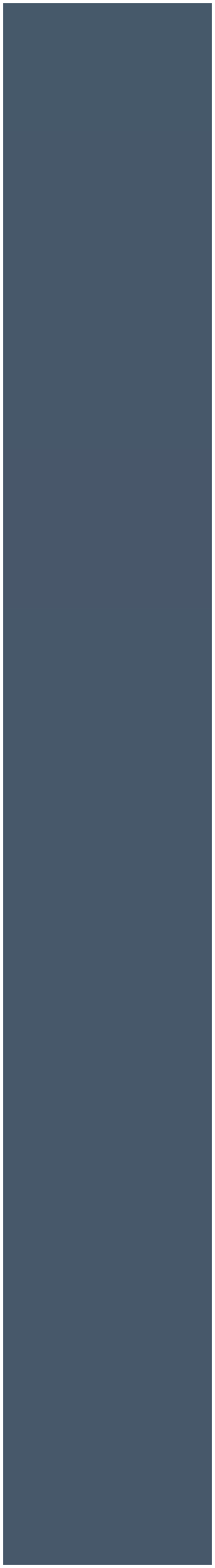
How to Build a Game with Vue.js

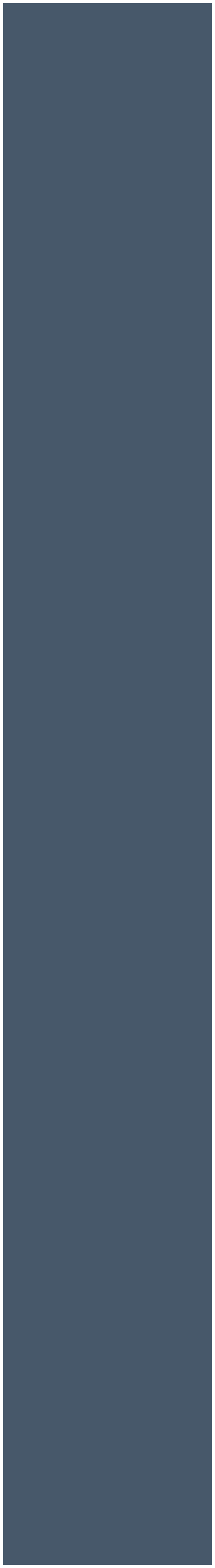


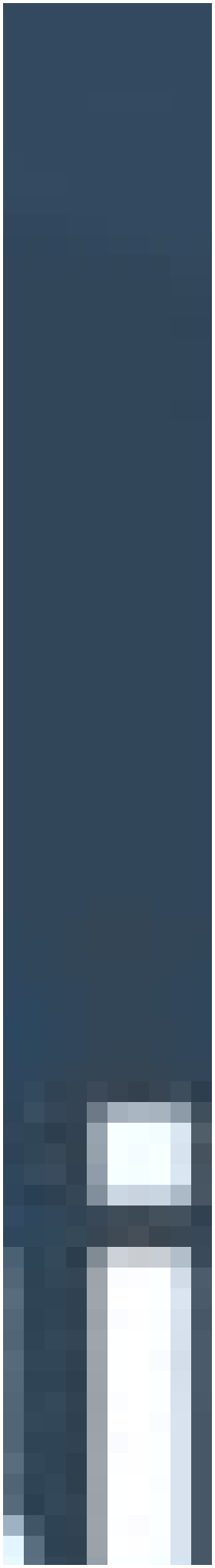


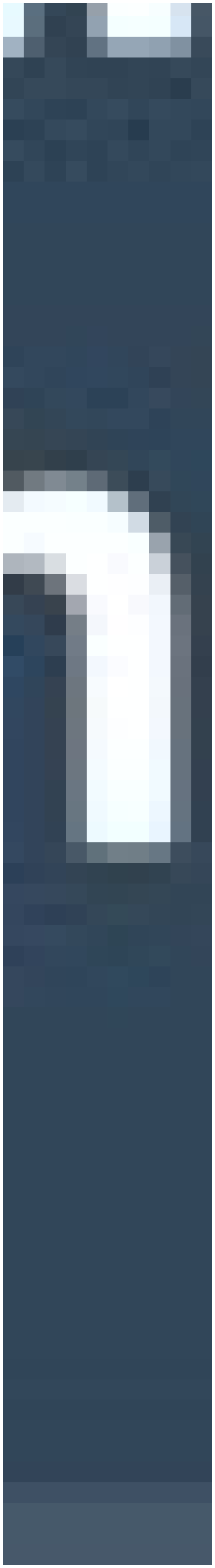




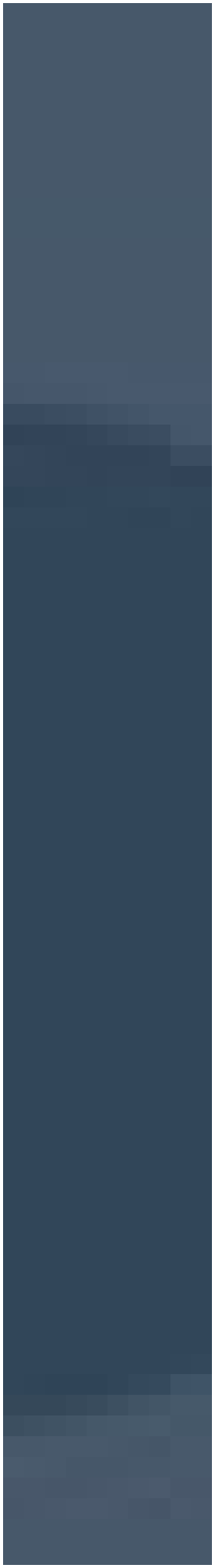


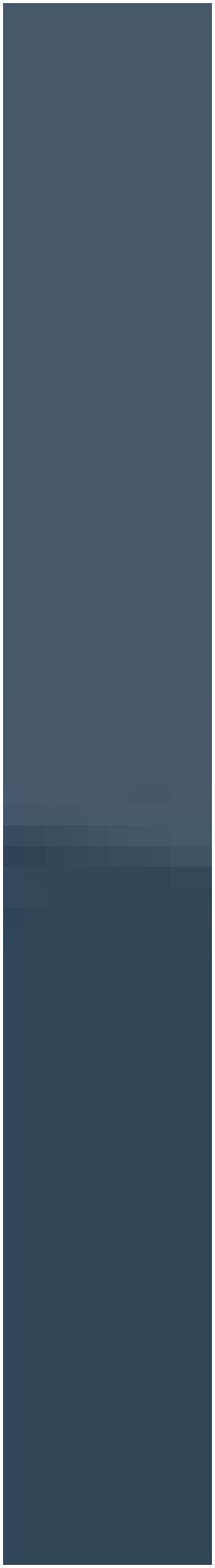






How to Build a Game with Vue.js





All JavaScript Books