2015年3月26日

株式会社イマジカデジタルスケープ

目次

[1. システムの構成について 4](#_Toc414917804)

[1.1 全体の構成 4](#_Toc414917805)

[1.2 サーバとクライアントの役割 5](#_Toc414917806)

[1.2.1 サーバの役割 5](#_Toc414917807)

[1.2.2 クライアント - コントローラの役割 5](#_Toc414917808)

[1.2.3 クライアント - ディスプレイの役割 5](#_Toc414917809)

[1.3 サーバの構成 6](#_Toc414917810)

[1.3.1 サーバ 6](#_Toc414917811)

[1.3.2 データベース 6](#_Toc414917812)

[1.3.3 その他ソフトウェア 6](#_Toc414917813)

[1.4 クライアントの構成 6](#_Toc414917814)

[1.4.1 コントローラ 7](#_Toc414917815)

[1.4.2 ディスプレイ 8](#_Toc414917816)

[2. 通信用APIの仕様 9](#_Toc414917817)

[2.1メタバイナリ 10](#_Toc414917818)

[2.2 リクエスト/レスポンスコマンド 10](#_Toc414917819)

[2.3 更新通知コマンド 11](#_Toc414917820)

[2.4 コマンド送信例 12](#_Toc414917821)

[2.4.1 画像登録の例 12](#_Toc414917822)

[3. データ構造について 13](#_Toc414917823)

[3.1 DBのデータ構 13](#_Toc414917824)

[3.2 データ格納方法について 14](#_Toc414917825)

[4. コントローラについて 15](#_Toc414917826)

[4.1 コンテンツの登録について 15](#_Toc414917827)

[4.2 仮想ディスプレイ全体の設定について 15](#_Toc414917828)

[4.3 コンテンツ/ディスプレイの設定について 15](#_Toc414917829)

[5. ディスプレイについて 17](#_Toc414917830)

[5.1 ディスプレイの登録について 17](#_Toc414917831)

[6. 画面操作について 17](#_Toc414917832)

[7. 動作環境について 18](#_Toc414917833)

[8. モジュール構成 18](#_Toc414917834)

# システムの構成について

## 1.1 全体の構成

本システムは, node.js及びwebsocketsを用いたクライアントサーバプログラムであり, 複数のアプリケーションや,複数のユーザが共同作業を行える, 巨大なスクリーンスペースを, 仮想ディスプレイとして提供する. 以下に構成図を示す.

## 1.2 サーバとクライアントの役割

### 1.2.1 サーバの役割

* HTTP - クライアントに対してHTMLページなどを送信する
* Socket.IO - node.jsによる通信機能で, クライアントのコントローラとの間で通信を行う.
* Websocket - websocketプロトコルにより, クライアントのコントローラ及びディスプレイとの間で通信を行う
* データベース - クライアントから送信された画像データやテキストデータをredisに保存し, 永続化する.
* サーバサイドレンダリング - 送信されたURLのウェブサイトをphantomjsを用いてサーバサイドでレンダリングし, png画像として保存する.

### 1.2.2 クライアント - コントローラの役割

* ディスプレイの登録, 削除
* ディスプレイの移動, 拡縮 などのメタデータ編集操作
* コンテンツ(画像データ, テキスト, URL)の登録, 削除
* コンテンツの移動, 拡縮 などのメタデータ編集操作
* ディスプレイの分割数の設定
* コンテンツデータのダウンロード

### 1.2.3 クライアント - ディスプレイの役割

* 自身のウィンドウをディスプレイとしてサーバへ登録する
* コントローラで設定されたとおりにコンテンツを表示する.

## 1.3 サーバの構成

### 1.3.1 サーバ

サーバは, node.jsを使用して, socket.io及びwebsocketによる通信に対応している. それぞれの通信方法によって受付ポートを分けている.

|  |  |  |  |
| --- | --- | --- | --- |
| 受付ポート | 通信方式 | クライアント | HTTPアクセス時 |
| ポートA | socket.io | socket.ioを用いたコントローラによって使用される | スタートページが表示される |
| ポートB | websocket | websocketを用いたディスプレイによって使用される |  |
| ポートC | websocket | websocketを用いたコントローラによって使用される |  |

### 1.3.2 データベース

データベースは, redisを使用して, 高速なレスポンスを実現している.

### 1.3.3 その他ソフトウェア

ウェブページレンダリング用にPhantomJS, 及びphantom.jsのnpmラッパーであるphantomjsを使用している.

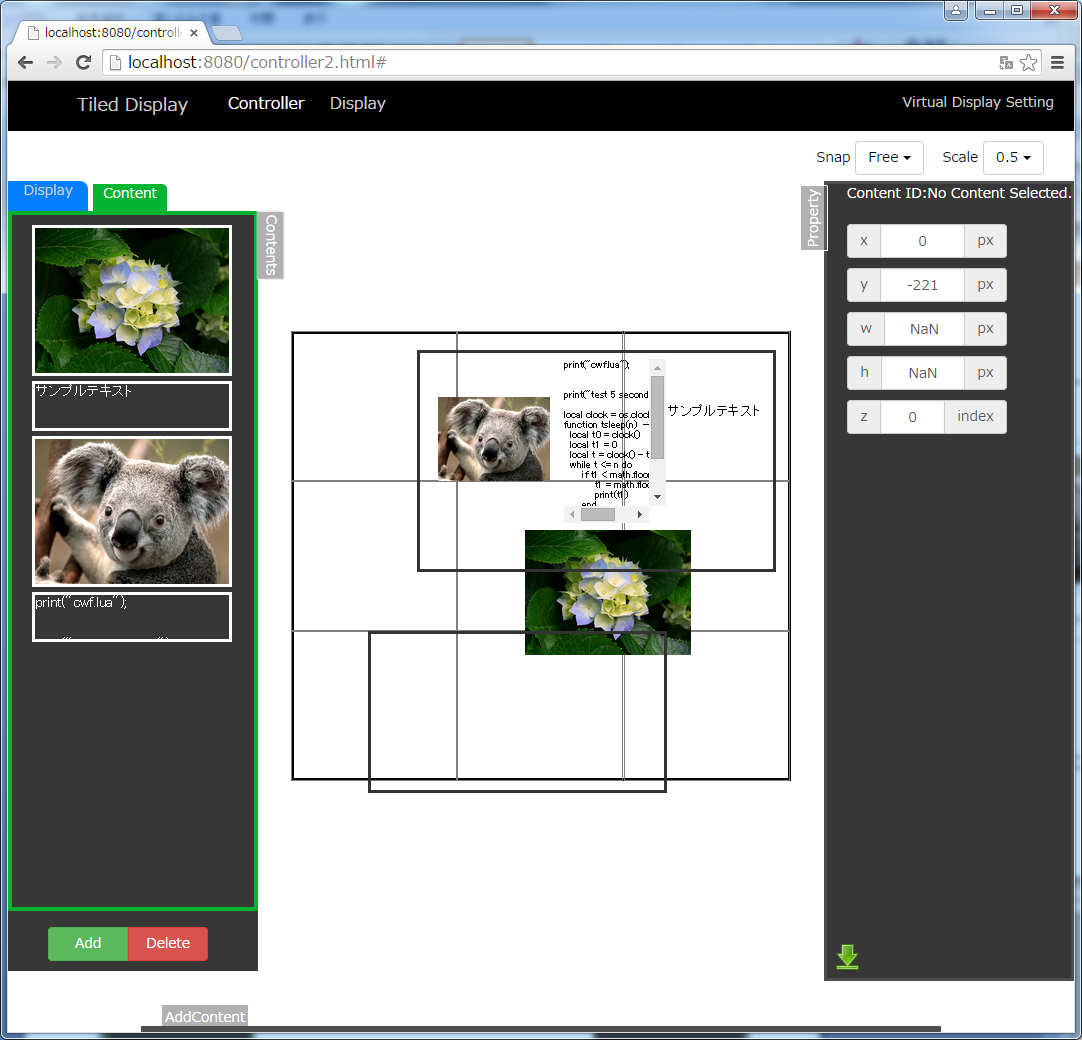
## 1.4 クライアントの構成

クライアントサイドは, 仮想ディスプレイに対してコンテンツの追加登録等の操作を行う「コントローラ」と, 表示のみを行う「ディスプレイ」から構成されている.

### 1.4.1 コントローラ

コントローラは, socket.ioを用いてサーバと通信を行っている.

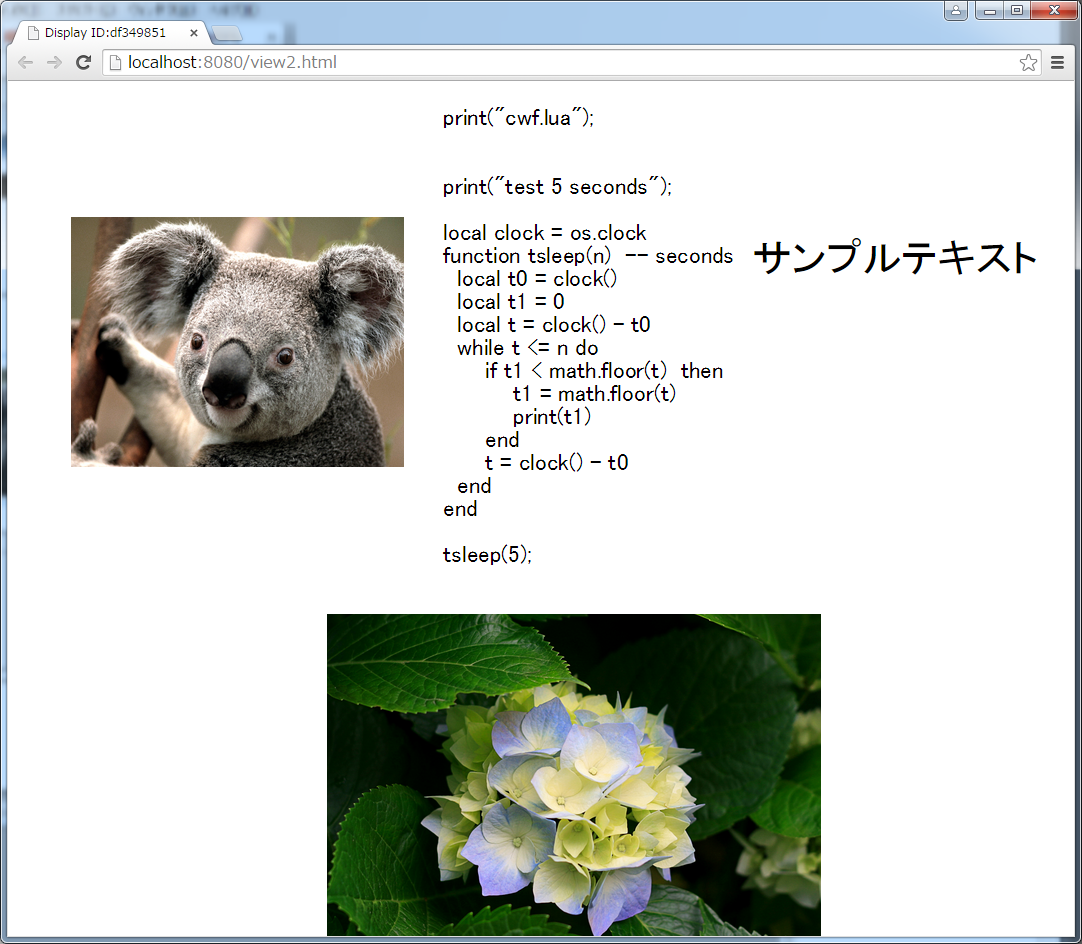
コントローラの画面イメージを以下に示す.



コントローラ

### 1.4.2 ディスプレイ

ディスプレイは, コントローラで設定したコンテンツを表示する. HTML5の機能であるwebsocketを用いてサーバと通信している. ディスプレイのイメージを以下に示す.



# 通信用APIの仕様

クライアントサーバ間の通信についての仕様を記載する.

サーバではクライアントからwebsocketまたはsocekt.ioでメタバイナリを受け取り, メタデータに記載されているコマンドによって処理を行う. 処理を実行した後, クライアントに対して, レスポンスを含んだメタバイナリをwebsocketまたはsoceket.ioにて送信する.

サーバ

ディスプレイ１(Client)

Websocketで更新通知をブロードキャスト

Socket.ioでリクエスト

ディスプレイ２(Client)

Socket.ioでレスポンス

コントローラ(Client)

コントローラからリクエストを送ったときの通信の流れ

データの更新が発生した際は, 更新通知をクライアントに対してブロードキャストする.

ディスプレイは, 更新通知を受け取ると, サーバへコンテンツ/ウィンドウ情報取得リクエストを送り, コンテンツ/ウィンドウ情報を取得する.

また, ディスプレイ新規表示時には, ディスプレイからサーバへ, ディスプレイ登録リクエストを発行し, 自身のディスプレイを登録する.

## 2.1メタバイナリ

クライアントサーバ間でやり取りするデータである, メタバイナリフォーマットのフォーマットを以下に示す.

|  |  |  |
| --- | --- | --- |
| ヘッダ | "MetaBin:" | 8byte |
| バージョン | UInt32 | 4byte |
| メタデータサイズ | UInt32 | 4byte |
| メタデータ | Ascii 文字列 (JSON) | メタデータサイズ byte |
| コンテンツデータ | Binaryなど | 残りのbyte |

コンテンツデータは, 具体的に以下の値が入る.

メタデータのtype=textの場合：UTF8文字列

メタデータのtype=urlの場合: URLエンコードされた文字列

メタデータのtype=imageの場合: 画像ファイルのバイナリ

バージョンは, 現在常に1が入る.

## 2.2 リクエスト/レスポンスコマンド

サーバで受け付けているコマンドの一覧を以下に示す.

|  |  |  |  |
| --- | --- | --- | --- |
| サーバへのリクエストコマンド | サーバからのレスポンスコマンド | 実行内容 | 更新通知 |
| reqAddContent | doneAddContent | コンテンツを追加します | update |
| reqGetContent | doneGetContent | コンテンツを追加します | 無し |
| reqGetMetaData | doneGetMetaData | メタデータを取得します | 無し |
| reqDeleteContent | doneDeleteContent | 登録されているコンテンツを削除します | update |
| reqUpdateContent | doneUpdateContent | コンテンツを更新します | updateTransform |
| reqUpdateTransform | doneUpdateTransform | コンテンツのメタデータを更新します | updateTransform |
| reqAddWindow | doneAddWindow | ウィンドウを追加します | updateWindow |
| reqDeleteWindow | doneDeleteWindow | ウィンドウを削除します | update |
| reqGetWindow | doneGetWindow | ウィンドウを取得します | 無し |
| reqUpdateWindow | doneUpdateWindow | ウィンドウのメタデータを更新します | updateWindow |
| reqUpdateVirtualDisplay | doneUpdateVirtualDisplay | 仮想ディスプレイのメタデータを更新します | updateWindow |
| reqGetVirtualDisplay | doneGetVirtualDisplay | 仮想ディスプレイのメタデータを取得します | 無し |
| reqShowWindowID | doneShowWindowID | ウィンドウIDを画面に表示するリクエストです。全てのDisplayに更新通知showWindowIDが送信されます。 | showWindowID |

## 2.3 更新通知コマンド

サーバから送信される更新通知コマンドの一覧を以下に示す.

|  |  |
| --- | --- |
| サーバからの更新通知コマンド | 通知内容 |
| update | 指定のIDまたは全てのコンテンツ/ウィンドウのデータが  更新されたことを通知します。 |
| updateTransform | 指定のＩＤまたは全てのコンテンツ/ウィンドウのメタデータが  更新されたことを通知します |
| updateWindow | 指定のIDまたは全てのウィンドウが、追加または  更新されたことを通知します。 |
| showWindowID | ウィンドウIDを表示する必要があることを通知します。 |

## 2.4 コマンド送信例

### 2.4.1 画像登録の例

まず, 登録するコマンドと, 幅や高さなどの情報を, メタデータとしてJSON形式で定義する.

metadata = {

"command": "reqAddContent",

"type": "image",

"posx": 100,

"posy": 200,

"width": 500,

"height": 400

};

このMetaDataと, 画像などのバイナリデータを組み合わせて,メタバイナリを作成する.

送信するメタバイナリ

Binary

MetaDataサイズ

MetaData

1

MetaBin:

コントローラから, サーバに, メタバイナリを送信することで, 画像が登録される. サーバでは登録時に,オリジナルのイメージサイズを, “orgWidth”, “orgHeight” としてメタデータに追加する. 登録が終了したら, “doneAddContent”コマンドを含んだ, メタデータがコントローラに返信される.

返信されるメタデータ

Metadata = {

"command": "reqAddContent",

"type": "image",

"posx": 100,

"posy": 200,

"width": 500,

"height": 400 ,

“orgWidth”: 500,

“orgHeight” : 400  
};

# データ構造について

## 3.1 DBのデータ構

本システムは, データベースとしてredisを使用しており, サーバによって受け付けたコンテンツやウィンドウ情報を保存している. 以下に, DBのデータ構造を示す.

|  |  |  |
| --- | --- | --- |
| キー | 内容 | 意味 |
| tiled\_server:s:default:virtual\_display | splitX | 仮想ディスプレイ全体のx方向分割数 |
|  | splitY | 仮想ディスプレイ全体のy方向分割数 |
|  | orgWidth | 仮想ディスプレイ全体の幅 |
|  | orgHeight | 仮想ディスプレイ全体の高さ |
| tiled\_server:s:default:content:[コンテンツID] | バイナリまたはテキストデータ | コンテンツの実データ |
| tiled\_server:s:default:metadata:[コンテンツID] | id | コンテンツID |
|  | type | コンテンツの種類 |
|  | posx | x座標 |
|  | posy | y座標 |
|  | width | 幅 |
|  | height | 高さ |
|  | orgWidth | 初期幅 |
|  | orgHeight | 初期高さ |
|  | zIndex | zインデックス |
|  | visible | 表示状態 |
|  | mime | mimeタイプ |
| tiled\_server:s:default:window:[ウィンドウID] | id | ウィンドウID |
|  | type | window |
|  | posx | x座標 |
|  | posy | y座標 |
|  | posx | x座標 |
|  | posy | y座標 |
|  | width | 幅 |
|  | height | 高さ |
|  | orgWidth | 初期幅 |
|  | orgHeight | 初期高さ |
|  | visible | 表示状態 |
| tiled\_server:s:tiled\_server:sessions | default | セッションID |

## 3.2 データ格納方法について

### 3.2.1 IDについて

コンテンツ情報, ウィンドウ情報は, それぞれコンテンツID, ウィンドウIDを割り当てて, データを格納している. IDはサーバでコンテンツ保存時に, ランダムな英数字8桁で作成される, “reqAddContent”などの追加コマンドに, 任意のIDを指定して追加することができる.

### 3.2.2 格納形式について

画像データ, テキストデータは, クライアントから送信されたものをそのままバイナリまたはUTF8文字列として保存している. URLについては, phantom.jsでレンダリングした画像データをバイナリとして保存している.

### 3.2.3 サーバで付与するメタデータについて

画像データについては, サーバ側で保存する際に, mimeを自動判別して保存している. また, phantomjsでレンダリングした画像については, posx, posy, width, height, orgWidth, orgHeightを, サーバ側で付与している.

# コントローラについて

コントローラページでは, コンテンツとディスプレイに対して各種操作を行うページである. 具体的には, コンテンツの登録, 削除, 移動, 拡大縮小, ディスプレイの削除, 移動, 拡大縮小, 仮想ディスプレイ全体の大きさの更新, 仮想ディスプレイ分割数の変更を行うことができる.

## 4.1 コンテンツの登録について

コントローラにて, 画像ファイル, テキスト, テキストファイル, URLをコンテンツとして登録することができる. 画像ファイルはjpg, gif, png, bmp形式に対応している. また, URLはサーバにてpng形式の画像としてレンダリングされて登録される.

|  |  |  |
| --- | --- | --- |
| コンテンツの種類 | 形式 | 備考 |
| テキスト | 文字列 |  |
| テキストファイル | txt |  |
| 画像 | jpg, gif, png, bmp |  |
| URL | png | サーバサイドレンダリング |

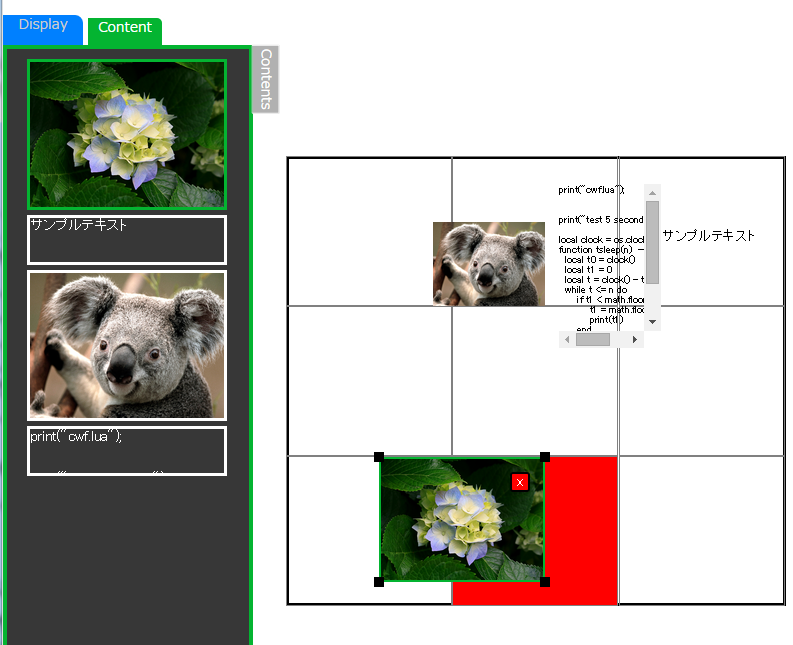
## 4.2 仮想ディスプレイ全体の設定について

仮想ディスプレイ全体の設定として, 仮想ディスプレイ全体の幅, 高さ, 分割数を設定することができる.

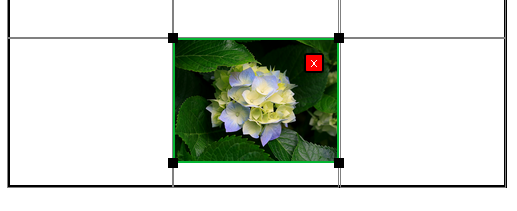
## 4.3 コンテンツ/ディスプレイの設定について

登録したコンテンツ及びディスプレイは, 仮想ディスプレイを表す矩形上に自由に配置し, 移動, 拡大縮小を行うことができる. また, スナップ設定を有効にすることで, 仮想ディスプレイの分割領域に対してフィットするように配置することができる.

分割した領域に対してスナップ配置しているイメージを以下に示す.



分割配置した領域に対してスナップ中の状態



スナップ後の状態

# ディスプレイについて

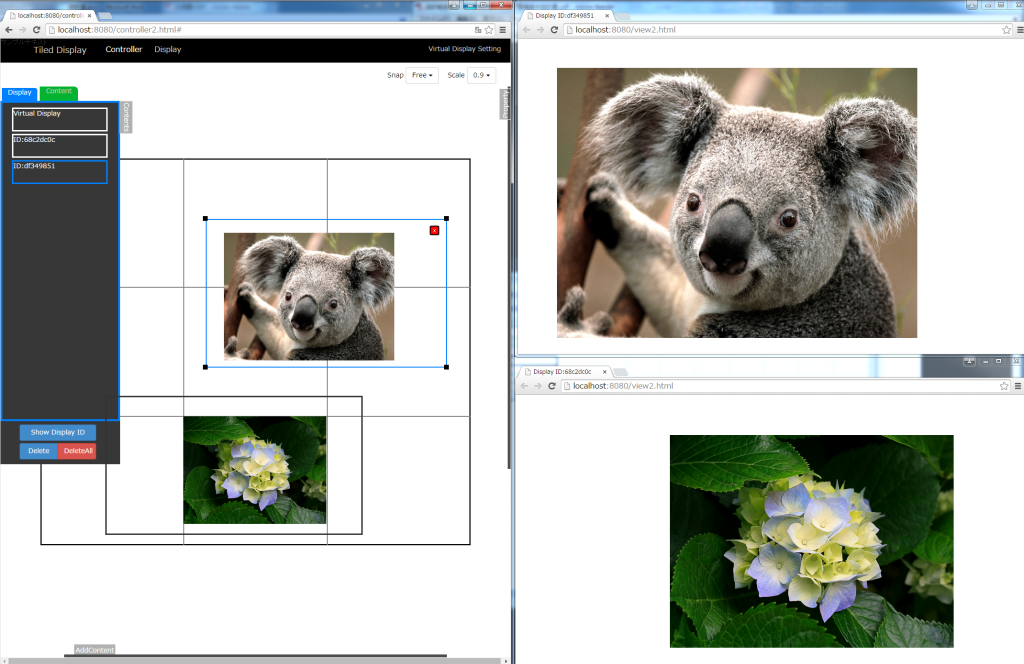
ディスプレイページでは, コントローラで操作した結果のコンテンツを表示することができるページである. 操作は全てコントローラより行う仕様となっている.

## 5.1 ディスプレイの登録について

サーバが起動した状態でディスプレイページを開くと, websocketのAPIを用いて自動的にサーバに登録される. 登録後は, コントローラにてコンテンツと同様に配置, 移動することができ, ディスプレイページを閉じると自動的に登録が解除される.

# 画面操作について

操作中の画面サンプルを以下に示す.



ディスプレイ操作のイメージ  
(左がコントローラ, 右がディスプレイ)

画面操作については, 別紙「利用説明書」を参照の事.

# 動作環境について

Windows, Linux, MacOSXのGoogleChrome, Firefox及びSafariで動作する..

各ブラウザの最新のバージョンにて動作確認を行う.

# モジュール構成

サーバサイド

* server/server.js - Websocket/Socket.io/HTTPのサーバ
* server/operator.js – DBへの読み書き, socket.io/websocketの全コマンドの処理を記述している.
* server/sender.js – websocketによるビュー用のコマンド送受信処理を記述している. 内部的にはoperator.jsのインスタンスを使用して処理している.
* server/util.js - ファイル入出力などのユーティティ
* server/metabinary.js - メタバイナリを作成するための処理
* server/command.js - 全てのコマンド名を定義している
* server/capture.js – phantom.js でURLをレンダリングするための処理を記述している

クライアントサイド

* client/js/controller2.js - コントローラの内部処理
* client/js/view2.js - ディスプレイの内部処理
* client/js/vscreen.js - 仮想スクリーン(全体, 個別)の矩形の処理をまとめたクラス
* client/js/vscreen\_util.js - 仮想スクリーンの矩形処理とmetadataに関するユーティリティ
* client/js/animtab.js - タブのアニメーションモジュール
* client/js/manipulator.js – コンテンツ/ウィンドウのマニピュレータの表示,削除,移動処理
* client/js/index.html – トップページ
* client/js/controller2.html – コントローラページ
* client/js/view2.html –ディスプレイページ