UbikTransmedia / **rvr**

Ruby Virtual Reality

MIT License

0 stars 0 forks

Star Watch

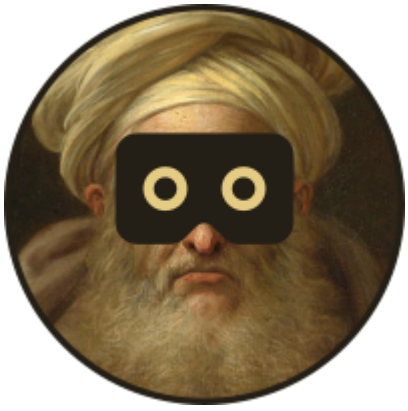
Code Issues Pull requests Actions Projects Security Insights

master Go to file

UbikTransmedia Active/Deactive ALL cargos. Each cargo has a unique... ... on Jun 27, 2019 21

View code

README.md



RiVeR 0.5

Ruby Virtual Reality: A standalone blueprint for cybernetic VR experiences

Hello WoVRd!

Welcome to Ruby Virtual Reality. This blueprint is meant to help you build cybernetic VR experiences, which means:

Flow > Graphics

Interacting > Watching

Connected > Lonely

As it progresses, the project aims for:

WebVR (A-frame) as the canvas

Concurrent users

Restful API

Asincronous queries (jQuery) in 3D

Runtime console/commands (Dungeon Master)

Controllers, controllers, controllers!

But give it some time! It just started! :) If feeling impatient, You can read more about the ethos behind this project at Jaron Lanier's books.

Install and run

RiVeR relies on Ruby and Sinatra to work. It is meant to be a blueprint, so no complex install, 3rd party sorfware nor surroinding apps are needed/provided. You get the files, and that's all. It works with a single command line and it will stay this way.

To run it: install ruby, download the files, get into the main folder through your favourite shell console, and type:

```
ruby rvr.rb
```

If you want to make the server public (so you can access it through any point of it's private network):

```
ruby rvr.rb -o 0.0.0.0 -p 4567 -env production
```

Then visit your IP or localhost and port 4567 to see:

```
http://localhost:4567
```

Basic concepts

RiVeR is, essentially, a collection of multiuser experiments; that you can borrow to speed up the development of you own projects. It's goal is to leap the gap from single to multiplayer user experience.

Sinatra legacy

Sinatra runs within the single file from which the library is called (*rvr.rb*(9): **require 'sinatra'**).

The *rvr.rb* file allows you to build web routes, this way:

```
get '/route-to-destiny' do
        # whatever you want to process
        erb :view_to_render
end
```

For more information about how Sinatra handles petitions, please refer to its documentation on routes and views.

Sinatra also uses a convention for resource handling. You will find the HTML templates at the */views* folder. Resources such as images, CSS and JavaScript are found (and fully available from anywhere within Sinatra) at */public* folder; for example, this is how you call a CSS file from the HTML templates at */views*:

```
<link rel="stylesheet" type="text/css" href="css/style-cargo-sender.css">
```

Rails legacy (ActiveRecord)

ActiveRecord is a object model for database handling. Instead of writting SQL sequences for the database, ActiveRecord turns every table into an object with easy methods. For more information, please refer to ActiveRecord's documentation.

Current database system is SQLite, a lightweigth database that fits in a file.

Please remember: ActiveRecord objects are translated as the plural of their table names. So a table called *users* will be accessed through a class called *User*. For example, in order to retrieve the user with ID equal to 5:

```
User.find(5)
```

You can extend any class to be handled as a database object like this (just note that you will have to extend the database; you can use the **DataBaseHelper.build** helper for this):

```
class Cargo < ActiveRecord::Base
end
```

Blocks

Commonly, frameworks attempt to comprise all the similar functions in conceptual structures (persistence, models, testing, etc.) so, once the architechture is clear, secondary apps can go through it seamslessly. This is the case of Rails, which provides a whole set of command line tools and development conventions to help projects get bigger (version control, TDD, database migrations, etc.)

In this case, RiVeR comes with a folder called **/blocks**, which is rather the opposite. Every implementation is meant to be as independent as possible from the rest; as simple as possible overall. There are no more conventions that Sinatra's and ActiveRecord's. The reason why is that RiVeR is not meant to be a full spectrum framework, but a launch ramp to help you skip the wikklitymikklity parts of development and focus on creating enticing WebVR experiences.

Current blocks are:

db_helper.rb

Makes database connection easy. You can edit its code in order to extend persistence. It's a Ruby module with three tools:

DataBaseHelper.setup

> Checks if database exists; otherwise, it creates it.

DataBaseHelper.connect

> It binds you application to a given SQLite file. You can also change the addapter and parametres (more at ActiveRecord adapter documentation).

DataBaseHelper.build

> If called, it creates the database and its tables. If you want your database to handle more types of data, this is the method you should extend.

db_models.rb

This file helds the class models that will inherit *ActiveRecord::Base*, so you can handle their tables in the database.

randos.rb

A collection of random generators. The current random function *Randos.test_cube*, for example, calculates a random 3D cube; if placed at the **return** of any AJAX function, it will send a test 3D cube into the scene.

A-Frame

A-Frame is a WebVR implementation that makes building 3D scenes easy. RiVeR imports it by default, and uses it as a way to boost develpment.

A-Frame works with HTML components in order to extend itself. For example, this wasd-controls component will allow you move using the keyboard:

```
<a-scene wasd-controls></a-scene>
```

A-Frame HTML code and components are a **really powerful** tool for WebVR development. RiVeR has been conceived to wrap all backend functions and let you focus on this. Please, do not underestimate them and refer to A-frame documentation, specifically about component creation. With some HTML, CSS and JS, you will be building multiplatform, interactive, connected 3D experiences.

> List of A-Frame ready-to-use components
>
> Detailed info on writting a component

Skills

Holodeck

The holodeck comprises a 3D space that users can visit, a storage of coded that has been loaded to that space, and ways to inject such code within the experience.

Code loads are called "cargos", as in the cargo cult. Master users send code to the database through the */cargo-sender* route. Later, a client-server relation is stablished with visitors so cargos can be delivered to foreign computers. The basic protocol works as follows:

1. Client declares the IDs of the cargos it already has (*public/js/cargo-loader.js*: **var cargoListing**).
2. The server compares the IDs from client with the IDs in the database. It will return a key-value listing with the proper IDs (*rvr.rb*(51): **cargo_delivery = {}**).
    i. If the ID is already in the database, the server assumes it is the right one and appends the key with an empty value.
    ii. If the ID is not in the database, the server will no append a thing to the **cargo_delivery**.
    iii. If an ID is in the database but has not been declared by the client, the server will assume that the client needs to add that code. Hence, it will send a key equal to the ID with a value equal to the code to be appended.
3. Client receives and discerns wether to keep, remove or update assets.
    i. If an ID is not in the **cargo_delivery**, it's DOM object will be removed.
    ii. If it is and the value is empty, it will be kept.
    iii. In case of receiving IDs with code, these code snippets will be added as DOM objects.

Note: currently, the delete function needs to be polished as cargos still do not have a class-id that let RiVeR delete them if unnecessary.

views/holodeck.erb

The real thing. The A-Frame scene that is constantly updated thanks to */public/js/cargo-loader.js*. It sends AJAX post queries **/cargos-update**, handles the answer and updated the DOM model.

views/cargo_sender.erb

An interface that posts data to **/cargo-sender**, which is inscribed within the *cargos* table of the *persistence.db* database.

Database structure

SQLite stores every **Cargo** object at the *cargos* table. You can handle Cargos anytime by following ActiveRecord's conventions; for example:

```
Cargo.find(1)
# returns a record object with the first cargo

Cargo.all
# returns an array with all the cargos

Cargo.create(
        code: params['cargo'].to_s,
        active: true,
        author: params['author'].to_s)
# creates a new cargo record in the database

Cargo.find(5).destroy
# will find the record with ID = 5 and delete it

Cargo.destroy_all(author: 'Donald Trump')
# will delete all the code snippets by Donald Trump
```

Agora

(To be implemented)

Agora will coordinate the sessions of many users, so they can stablish user-to-user connections for sharing realtime data through WebRTC. This will make multiuser experiences available.

Progress log

> Holodeck is working: you load code in a database, and code is passed to the 3D scene.
> Holodeck needs to fix the way it destroys DOM cargos that should not be there.
> Current: polishing holodeck + adding holocked skills.
> Next: [new demo]: inboun API to scene.
> Pending: multiuser scene (WebRTC).

But give it some time. It just started! :D

RVR 0.5 - Guillem Carbonell - g@ubik.bz - 2019

Releases

No releases published

Packages

No packages published

Languages