Mark Dolbyrev For Game Closure

Guild Of Wicked

Follow-up for Code Challenge by Mark Dolbyrev.

Web App: https://deletris-static.herokuapp.com/guildofwicked/index.html
Sources: https://github.com/flashist/guildofwicked

Guild Of Wicked

Introduction

Interesting & time-consuming

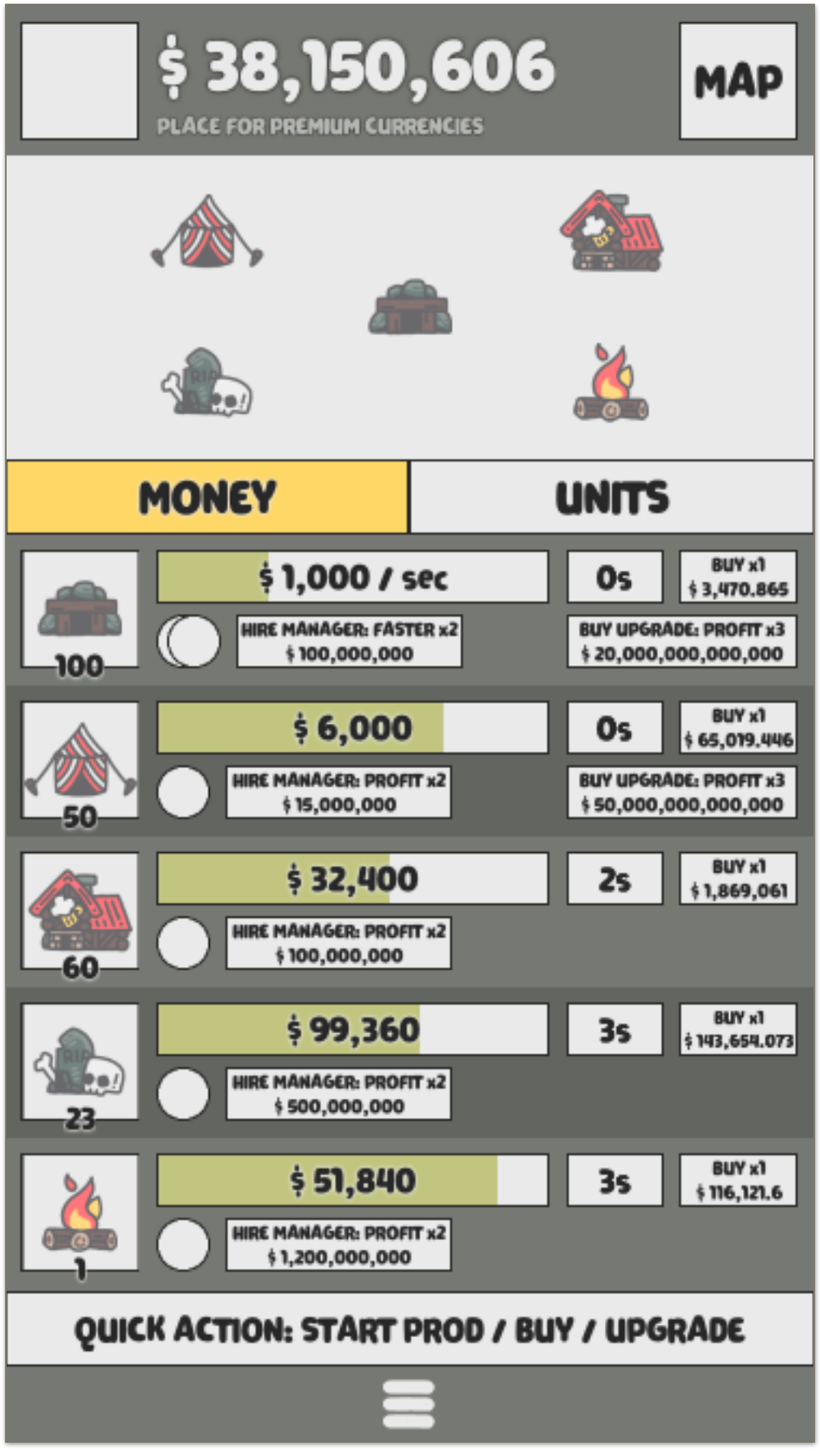
In this document I'll try to explain my working routine, taking the Code Challenge task as an example. I'll try to focus on different aspects of work to give a full picture of how I work on projects and why I made some decisions in this very specific game-prototype.

The main areas of work include:

- Client-Side Development
- Server-Side Emulation
- Game Design
- Art

I'm glad that I had a chance to participate in the Code Challenge, though I definitely went beyond of supposed time limits.

It was very interesting and so far I like some software solutions I prepared while working on this task.



Spec Requirements

The main spec requirement points are mentioned at the beginning, because I value your time:

- **Buy and upgrade businesses**
  The 1st business is given for free, other businesses should be bought by a player.

- **Make money from a business**
  Click on the icon of a business to start production.

- **Hire managers, so that money is made automatically**
  As in Adventure Capitalist it happens, if you buy 1st level managers for each building.

- **If something is produced while you're offline, you'll see a notification about it at the next login time.**
  Implemented!

- **The implementation should have several business types to choose from**
  The game has 8 types of businesses (generators), which produce 2 types of resources (money, attack).

- **Must be built in a dialect of JavaScript: ES5, ES6, ESNext, TypeScript, etc.**
  GIT + TypeScript v3.8.3 + Pixi.JS v5.2.1 + Howler v2.1.3 + Webpack v4.42.0 + CreateJS (for loading)

- **The UI can be anything: Visual, textual, even a fullscreen CLI works.**
  I decided to create art-mockups for the main pages of the game (preloader, game-production, game-map).

- **Visual Polish**
  Art is not my area of expertise, but I decided to create a workable art-mockup for the app to show the main elements of UI and the game.

- **Extra Features**
  The game has different type of resources and generators (buildings) for producing them. The game is built fill all the available screen size. Many features are not implemented but described in the Game Design section (e.g. world map, conquering locations, war units).

- **Server component:**
  The current implementation of the server-emulation can't be called as real-use case server-side implementation, but it represent the main server-side logic, client-server protocol, recieving requests from client-side, saving data for keeping users progress.

- **Write your README**
  Done. Added to the repo and email.

Development: Client-Side

From the beginning, I decided that the final app won't be just a prototype which covers the main requirement points from instruction, even though it might be an overkill for the Code Challenge. This decision is based on the next reasons:

- Knowing that I wouldn't have much free time to work on the project, because at my main job we've had pretty busy weeks before an important upcoming milestone, I just didn't want to spend my time on something I don't like. I decided to work on something which can be reused in future, to make sure the code can be reused as a boilerplate for future applications.
- I wanted to present some architectural solutions which, in my opinion, can be used for complex games and applications and go far beyond limits of small prototypes.
- I wanted to be able to discuss and defend the solutions I used without such excuses like "it's just a prototype".

Taking all the above into account, I decided to use my own libraries, which has been developing for quite some time and reused as a basement for my pet-project applications:

- **fcore** (https://github.com/flashist/fcore): low level library, not connected to graphics applications.
- **fsuite** (https://github.com/flashist/fsuite): a) Graphics / Sound library independent lib (currently only pixi.js + howler are supported, but the lib was built in a way to add support of new graphics/sounds libs easily). b) Some custom solutions, which can be used in a variety of projects (e.g. Service Locator, Object Pools, Loading Files, Localization, Working with HTML, etc.)
- **fconsole** (https://github.com/flashist/fconsole): a debug utility for display-list based applications. Sort of a port of a very useful debug tool Flash Console: https://github.com/junkbyte/flash-console
- **App Framework**: currently not moved into a separate lib, as is still in active development state. Located under the src/appframework folder of the game. The App Framework is meant to be a boilerplate for further apps development and used as a software-architecture guid line.

Configuration Files

The game is built on top of some of my libs and architectural frameworks. In this architectural paradigm specific structure of config files for loading assets, localization and initialization is required:

- **assets/**
  All configuration files and assets (index.html is the only exception) are located under the assets/ folder.

- **app-config.json**
  The first file downloaded by apps. Consists of links to other config files (assets.json, texts.json, static.json) and additional app settings (e.g. sizeArea).

- **assets.json**
  Configuration of files to be loaded. The order of loading might be changed by assigning different loading groups to files and changing their position in the list of file. Setting loading group settings to assets can be used for minimizing time of showing “empty” screen to users (until the preloader page is shown), because as soon as all assets under the **preload** group are loaded the client-side app knows that the preloader page can be initialized and other assets won’t be waited to be loaded. Also, grouping files can be used to create lazy-loading features (e.g. additional effects, sounds, feature-related assets) or loading-on-demand assets.

- **texts.json**
  List of texts used in app. Different files should be used for different languages.

- **static.json**
  Configuration of app static data. In the current architectural approach data in apps can be separated into 2 different groups **static** and **dynamic.** Static data is the data which is never changed (e.g. generator prices, resource production settings, bonuses). Usually this data is downloaded at the beginning.

In real-life examples the paths to the files might be configured by placeholders to use different paths to different language-related assets or DPI-related images. For the purpose of minimizing time efforts only localization placeholder is used in the Code Challenge prototype (see path to the texts.json file).

Initialization And Loading

- **index.ts**
  The entry-point of the game is the index.ts file. The only purpose of it to initialize the game-specific instance of the **Facade** class.

- **Facade**
  Facade class is a singleton class which is responsible for the initialization of an app, creation main visual elements, activating modules and main components of the game.

  In this architectural paradigm the game consists of separated blocks of modules. One of the main responsibility of the Facade class is to add needed “blocks” (**Modules** would be discussed more later)

- **GOWFacade**
  App-specific implementation of the Facade class. The main purpose of the app-specific Facade class is to activate app-specific modules, which are responsible for app-specific logic of apps.

- **InitApplicationCommand**
  The first command executed by apps. Consists of a list of commands which are responsible for initialization if the app. It waits for the preloader page assets to be loaded and shows the preloader page.

- **LoadAppConfigCommand**
  Loads and parses the app-config file.

- **LoadLocalizationCommand**
  Loads and parses the localization config file.

- **LoadStaticItemsCommand**
  Loads and parses the static-configuration file.

- **InitLoadProcessCommand**
  List of commands responsible for loading the assets configuration file and initialization of the loading process.

Modules

Modules are the main building blocks of the apps in the current architectural paradigm. Modules can be thought as a box of classes which are responsible for some functionality. In a perfect world all modules would be independent, but in real-life cases in 99% of production apps at least some of modules depend on other modules.

Let's take a look in the **appframework/app/** folder and the **AppModule** class:

- The app module consists of different type of components, which are combined in folders according to their types (e.g. commands, data, managers, models, views).

- The AppModule class is responsible for configuration of the classes, which require configuration (e.g. models and managers are usually configured as singletones). Configuration of other classes are done in the **init** method and usually consists of passing specific configurations into the **ServiceLocator** class (**ServiceLocator** will be discussed more detailed later).

- Current architecture supposes that appframework-level classes can be changed to implement app-specific logic. Changing appframework-specific classes are done through substituting them by using **ServerLocator** functionality (**ServiceLocator** will be discussed more detailed later). Substitution of appframework-specific classes by app-specific classes is usually done in app-specific modules (see the **GOWAppModule** class).

- The current architectural framework is built to be flexible for further changes and use-cases which are not defined at the current moment. For this purpose **BaseModule** class has a couple of hook-methods, which are called when **initialization** and **activation** of modules are done.

- **Activation Hook** is needed in cases when some actions should be done after initialization of modules (e.g. configuring page classes, which are responsible for showing different pages, see the **GOWGameModule** class).

Service Locator

Even though the Service Locator pattern is thought by some people as an example of anti-pattern, I found it to be very useful, convenient, flexible and giving a way to extend logic of future apps without knowing specific requirements today.

The main benefit of the Service Locator class in the current architecture is it's configurability and a way to change low-level-modules logic without overriding low-level-classes.

Let's take a look at the **GOWAppModule** class:

- In the init method we can see that the app changes the default app config class (**DefaultAppConfigVO**) to app-specific class (**GOWDefaultAppConfigVO**). This means that all other classes which will try to get an instance of the **DefaultAppConfigVO** class would get later an instance of the **GOWDefaultAppConfigVO** class.

- Another example in the same module is substitution of the **AppMainContainer** class by the app-specific **GOWAppMainContainer** class, which is done for extending low-level class logic and adding additional component as hints and containers management (see **GOWAppMainContainer**).

Usually objects which are got through the ServiceLocator class are created on-demand (which means, that they won't be created until some other code is trying to get them). In some cases it might be needed to create instances of objects as soon as the app as initialized. In this case classes can be configured with the flag **forceCreation** (see the **AppModule** class).

Current implementation of the **Service Locator class** gives us a way to configure the type of creation (e.g. singletone/non-singleton (**isSingleton**), creation on-demand/force creation (**forceCreation**)), substitution of other previously configured classes (**toSubstitute**), creation classes, when other classes are created (**activateeConstructors**). The last case will be discussed more detailed later on the example of **View + Mediator** classes.

Getting objects through Service Locator might be though as an example of **OOP Abstraction/Polymorphism**, because as long as interfaces of the objects are the same, low-level classes don't care with which instances they are working (app-specific or framework-specific).

Another real life example is using Service Locator mechanism to create server-emulation logic, by substituting real server-communication classes and returning some predefined answers, which might be useful if client-side app is developed faster than the server-side.

MVC

(M) Models + Value Objects
(V) Views + Mediators
(C) Managers + Commands

The current framework uses the MVC pattern as a basement. If we google explanation of the MVC pattern, there would be at least 2 different approaches to it:

1. Views can't read data from models. Only controller can read and pass the data to View classes.
2. Views can read data from models.

I prefer the 2nd approach as it allows us to write less code than the 1st one.

Models

Models present containers of data which can be used everywhere in the app. Data in models is presented as a model param and/or a list of VO items. Changing of data in models cases changes in the app.

Some models might be called **ViewModels** as created for view-related data only.

Views + Mediators

In the current architecture I think about Mediator classes as elements which connect **Views** with "outer world". In a classical **MVC** approach Controllers work with Views as black-boxes knowing nothing about their inner structure. In the current architectural approaches I prefer not to be so strict about **Mediators** and give them a way to know the inner structure of views to reduce the amount of written code. E.g. a **Mediator** might know a **Button** inside of a **View** class and listens the **CLICK** event from the button directly, without creating additional code of handling button events in the **View** class and then dispatching another custom event to the **Mediator** class.

Connection between a View and Mediator is implemented by using **Service Locator** mechanism, when Mediator class is created after creating View class.

Managers + Commands

Managers and Commands represents a control-layer of MVC pattern. Usually they don't have links to Views, but might listen to events from outside to perform some actions. Also managers might be used from the **ServerLocator** mechanism and their methods might be called directly (e.g. sending server requests).

Command, in general, my favorite pattern because it's as simple as it could be and in my opinion represent a perfect example of **OOP Encapsulation**.

Client-Server Protocol

One of the pros of the current architecture is the fact that client-server protocol has built-in rules, following which code responsible for client-server logic, might be drastically reduced.

For example: imagine if we have a game-feature of upgrading buildings. Originally, the upgrade gives some bonus for the building only. But imagine that we want to extend this logic and give some global bonus for each x100 building upgrades.

In regular approach it would mean that client-server protocol should be changed and server-side should add additional fields, which would tell the client-side about the global bonuses.

In the current architecture it can be done without changing client-server protocol (though work of showing changes would still be required):

- Client-side expects responses from the server-side to follow a basic protocol interface described in the **IServerResponseVO** class. The most interesting part of it is the optional **items** param, which represents a list of changed data items.
- In the example of buying upgrades and giving global bonuses the Server-Side app would send a list of global bonus objects, which are given to the user as a result of their actions (buying upgrades).
- If in future we would like to extend the logic and give some clothes or open additional buildings based on upgrades bought, server-side would add additional items to the list and the client side would now about the changes.
- Objects in the **items** list should follow a basic interface, which can be seen in **IGenericObjectVO**. The main params are **id** and **type**. Which would be responsible for finding needed data from all the items which server-side sends to the client side.
- In the example of bonuses, the server side might send to the client side the object **{type: “bonus”, type: “globalBonusId”}**, which would mean that the user now has a new bonus.
- In some cases server-side might send a command to remove an item from the client side (e.g. a bonus is taken away from a user). In this case the object might look like: **{type: “bonus”, type: “globalBonusId”, action: “remove”}**

- The current architecture is built to reduce amount of written code for common parsing operations. Parsing of items is automized too. Automatization is implemented by connected the **GenericObjectsByTypeModel** and **BaseServerCommand** classes, when the command class passes the items object into the model class to be parsed.

  The model class would go through the object and based on them either create new VO objects in models or update the existing ones.

- All parsed items can be got from the **GenericObjectsByTypeModel** using the **getItem** method.

- In some cases we might want to keep all the items in specific models (not the generic one). E.g. we might want to keep all information about users under the **UsersModel** class. That's possible too and done in the app-specific **GOWUsersModule** class in the activateCompleteHook method, where the **GOWUsersMode**l class is mapped to the specific item types to be parsed in this specific model.

- Sometimes we might create specific VO classes for specific items passed from the server side (e.g. server-side passes raw list of resources, but we want to provide a way for the client-side code to get only resources of a specific type). This can be easily implemented too. Check the GOWUsersModel class where a constructor class for the VO objects is specified.

- Another useful example can be found in the **GOWUsersModel** class too: the model provides a way to get the current user data directly, without passing the id of the current user (because the current user data is supposed to be used much more often than other users data).

- In addition to reducing code on the client-side the same approach might be used to simplify or reuse common solutions on the server-side. For example we want to implement a feature of buying new buildings. Usually it's done by creating a new request type and creation a code which is responsible for it.

  In the current architectural approach, a general request for buying objects might be defined. Which mean that the client-side would send only Type and Id of an object the client side wants to buy, without creation new requests.

  This is exactly what was done in the current prototype app. Implementation of buying managers and upgrades didn't require additional request methods. The general method for buying was used instead.

Development: Server-Side

Even though I've done some server-side development in my life (e.g. PHP for websites, Ruby for some game-related server-side code, Smartfox + Java for Flash-based applications), I've never considered myself as a full-stack developer. But I'm not afraid of doing it either, though I understand that there are some server-side specific things which I'm not aware of (e.g. optimization of working with Data Bases).

In the current app, I decided to create a Server Emulator class, which would represent API for client-side communication, while not being actually a server-side code.

Instead of using data bases or saving in local files, I used Local Storage, to emulate saving data on the server-side.

It means that user progress is saved in the browser and if anybody wants to start the game again, they might just clear their local storage for this browser tab.

As static files with settings of buildings, upgrades, bonuses have the same data which should be used on the client and server-sides, I decided to reuse some of the client-side interfaces and classes in the server emulation logic.

In real life examples, sharing of similar classes would probably be avoided, to prevent mix of client and server side logic.

In the middle of server-side development I started thinking that in real-life examples my server-side code would be more similar to the client-side code, with separation to Models + Value Objects + Managers.

I decided to keep the emulation-part of the app as simple as possible, so fewer classes with shared logic were created (e.g. manager classes combine models and controllers logic). It allowed me to reduce time required for development of the server-emulation classes and keep the code clean and easy for understanding and review.

Game Design

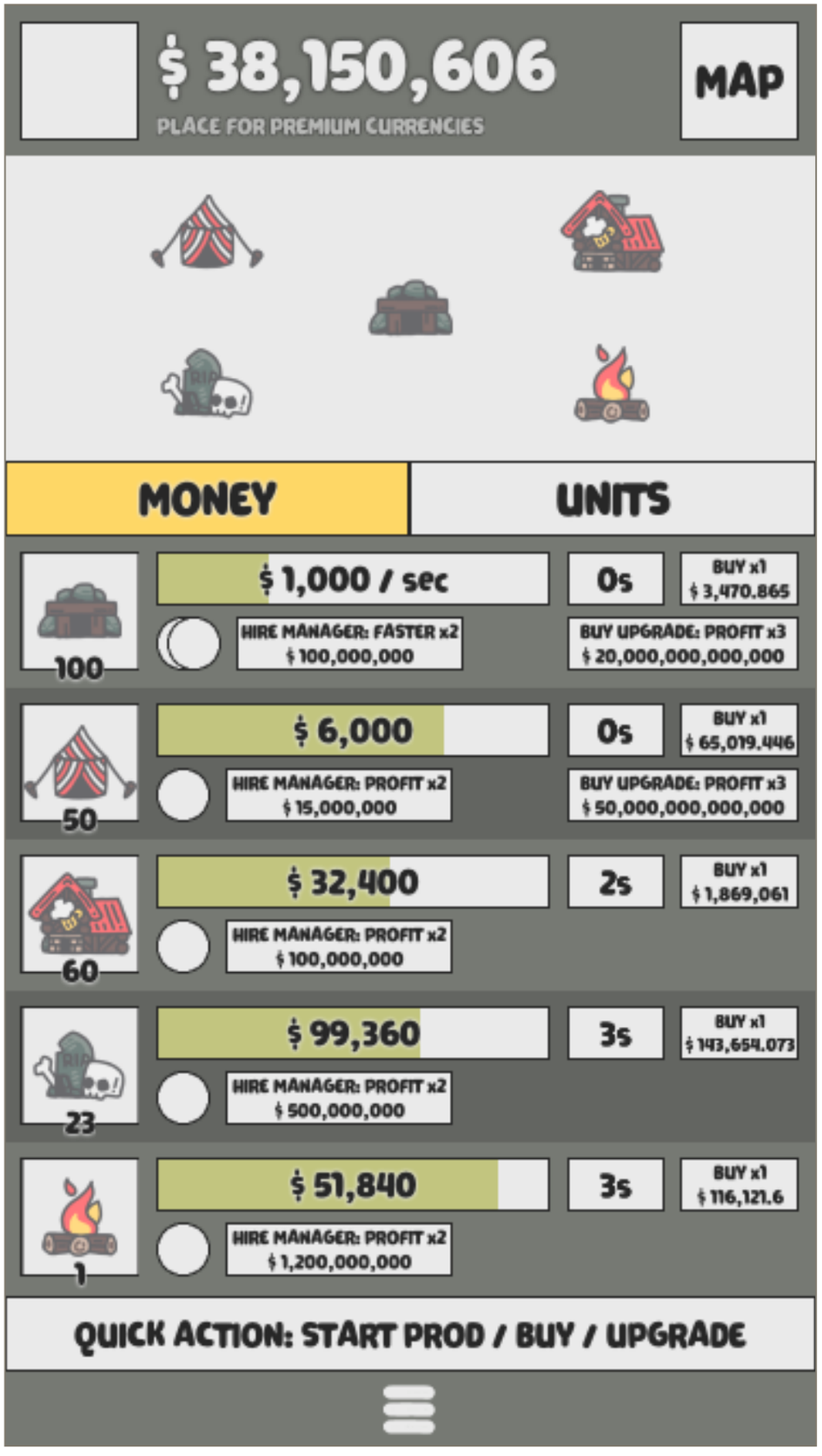
As was mentioned in the beginning, I knew that I wouldn't have much free time for the project. So from the very beginning I decided to create something I'm interested in.

I'm not very interested in the idler/clicker games in general, and in Adventure Capitalist game specifically, so I decided to brainstorm ideas for a game I would love to develop myself not only as a prototype for the Code Challenge.

At the beginning I found information about math behind clicker/idler games, which was very interesting and new for me. And while busy at work, I thought about different ideas in free time (e.g. commuting, before sleep), and the next idea came into my mind:

Clicker/Idler mechanic is a very simplified version of the economy layer of strategy games.

Which leads to the another idea: we can create a simplified strategy game, by adding other layers of strategy games to the clicker/idler games.



Units

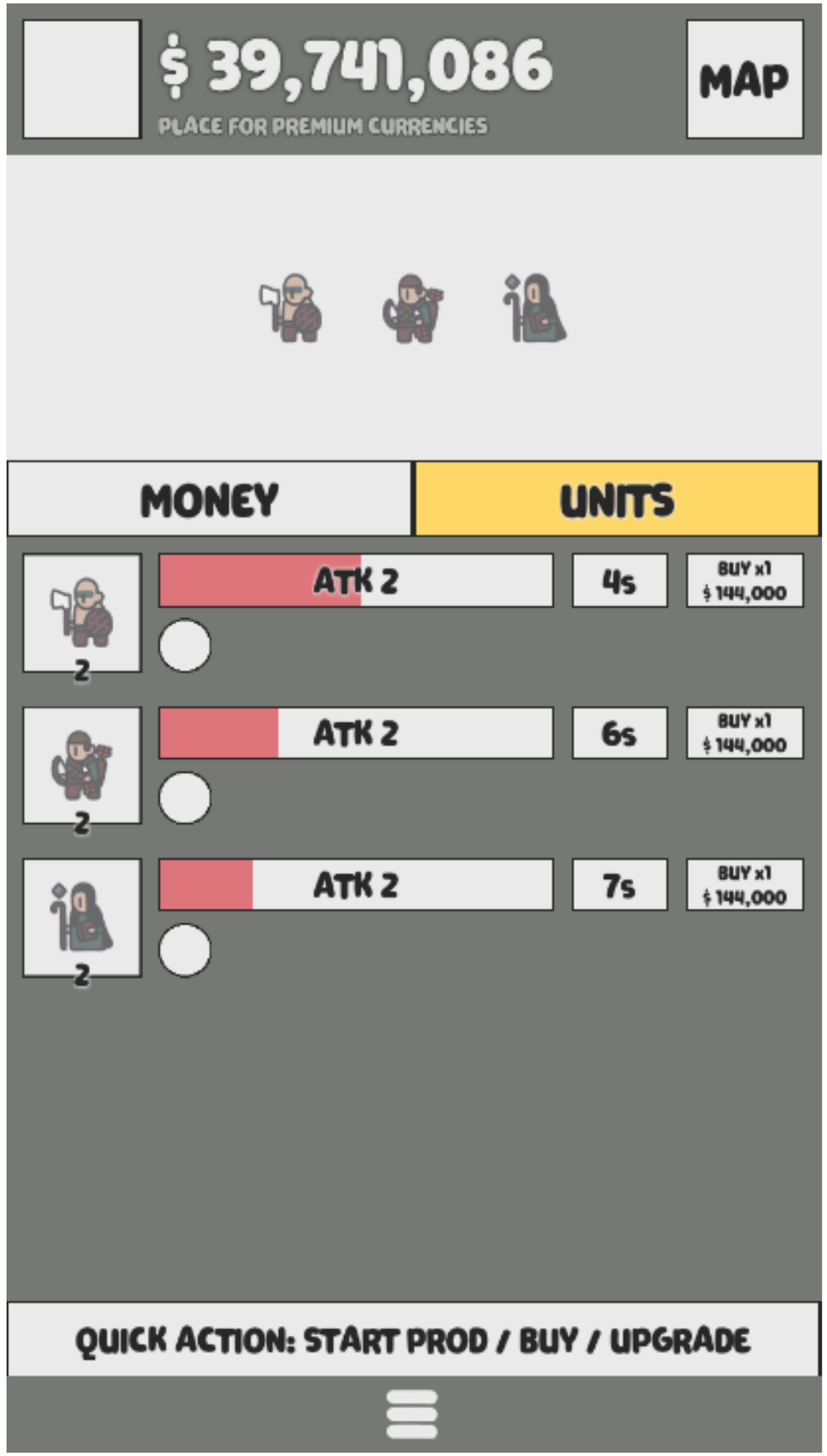
Units can be thought as generators, which produce other type of resources, e.g. ATTACK or DEFENSE.

The same as money-generators they can be upgraded, managers can be hired, more buildings can be bought, to produce more resources and automatize the production cycles.

But what next?

The main thing I don't like about clicker/idle games is that I got bored of them from some point (when you opened all or most of all buildings / upgrades / achievements).

In order to prevent the same "stuck" with the game, the ATK / DEF resources should be somehow spent. And that's how the idea of the **MAP** appeared.



Map

I like the “exploration” part of some games. Don’t know exactly why, probably it’s a feeling of finding something new. Good examples of exploration for me is Stellaris, Civilization: Beyond Earth and Curious Expedition.

I like the exploration part so much that in case of Stellaris and Civilization I’m usually not interested in finishing games, if there is nothing more to explore.

Taking all the above into account, the idea of Map appeared:

- When you’re developed enough in generators, you spend your money for units generators.
- When you have enough ATK resources, you spend them for conquering new locations. Some locations might give you additional one-time bonuses (e.g. artifacts) or global modifiers (e.g. gold mine) while you’re controlling the location.
- The more locations you conquered, the more locations you can see at the map.
- At some points you even might find other players, and that’s how player interactions start.

Game Design

Player Interactions

Sooner or later you find other players and they want to expand their borders as well as you! It means that you'll compete for locations on the map, especially for locations with bonuses.

It adds another reasons to develop your generators even after opening all of them: there will always be somebody who wants to conquer your territories or there will always be some locations you want to conquer.

Travel takes time and the further the location from your city, the longer it would take to go there.

New type of buildings come into play: buildings which gives your somethings, but not generate anything.

- Fort - improves your defense.
- War Banner - gives a way to combine more units in a squad.
- Stabbles / Transport Company - improves map travel time.

Game Design

Tactics
Rock-Scissors-Paper

As you have 3 unit generators, you would have 3 different squads to send for conquering tiles.

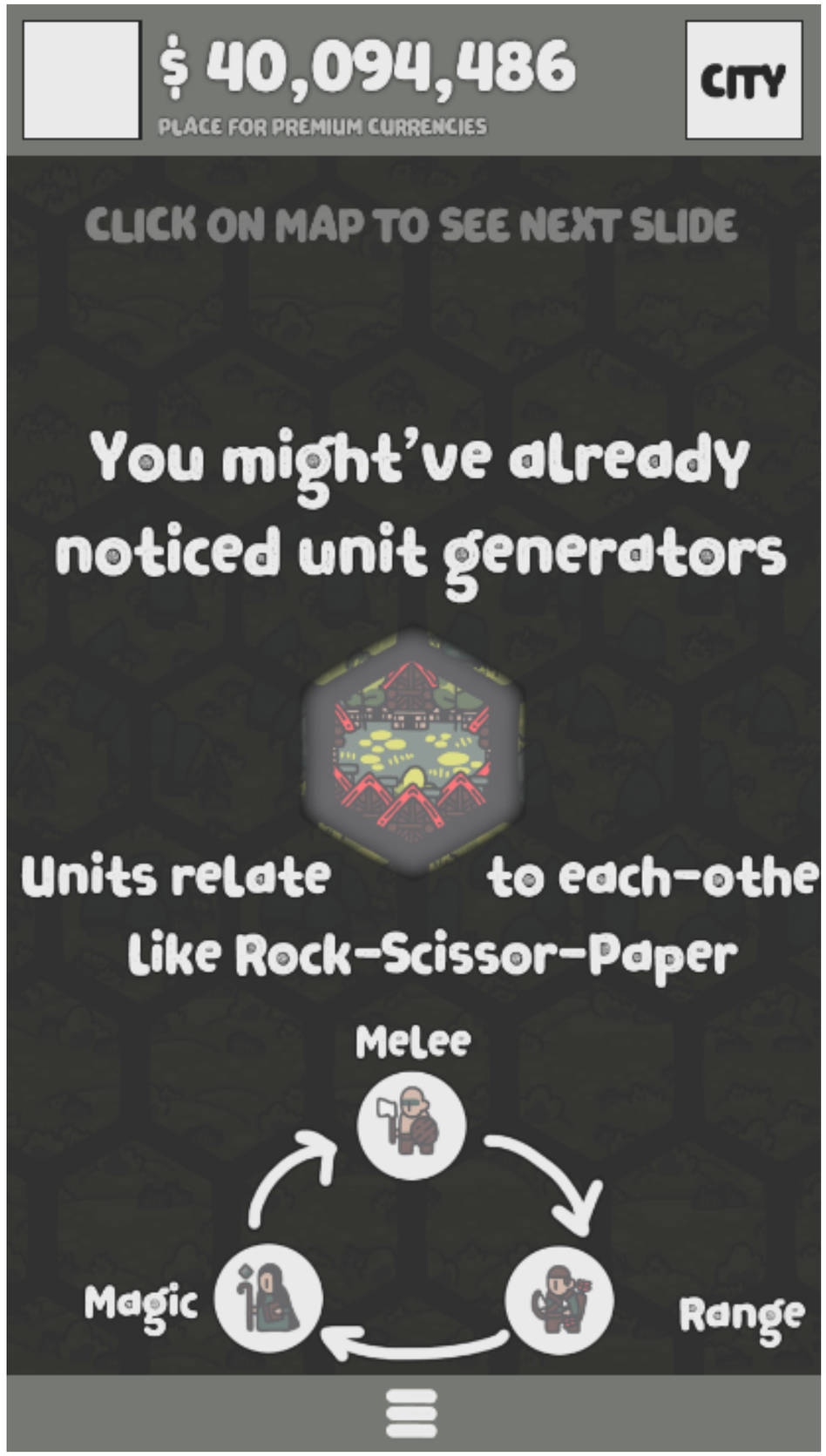
When a tiles is free, it means that NPC players control them, and it doesn't matter which type of units you would send for conquering.

When a tile is conquered, the type of units, which are sent for conquering the tile would be responsible for defending it.

The amount of "defense" in a tile is defined by the level of the Fort building of yours.

Different type of units relate to each other in a Rock-Scissors-Paper style: using one units against others is more efficient (e.g. melee units have advantage over Ranged units, but get more damage from Magic units). It means that you can conquer territories of stronger players by smarter decisions.

If a tile is attacked, but the damage is not big enough for conquering it, the defense of the tile is reduced. The defense of the tile is regenerated over time (based on the production level of the unit type, who is responsible for defending it)



Gameplay Cycles

- **Cycle Of Generation**
  Start Production > Wait

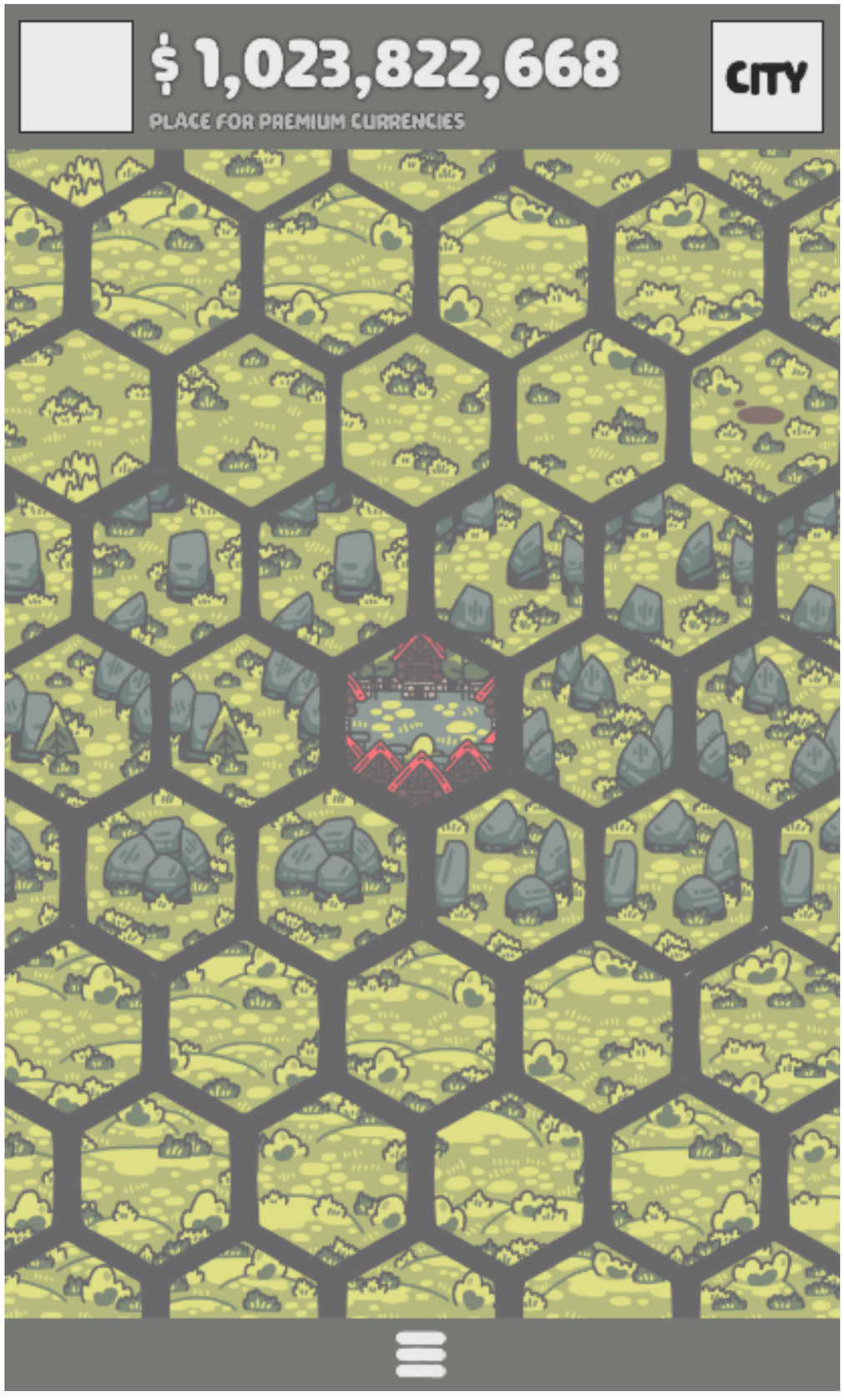
- Cycle Of Upgrade
  Get Resources > Upgrade

- **Cycle Of Conquering**
  Get Units > Conquer Location > Gather Bonuses

- **Cycle Of Defending**
  Conquer Location > Defend Location

- **Cycle Of Exploration**
  Choose Location To Conquer > Conquer It > New Locations Open

New layers of gameplay and player interactions makes the game more interesting to play and adds new ways for monetization.



Guild Of Wicked

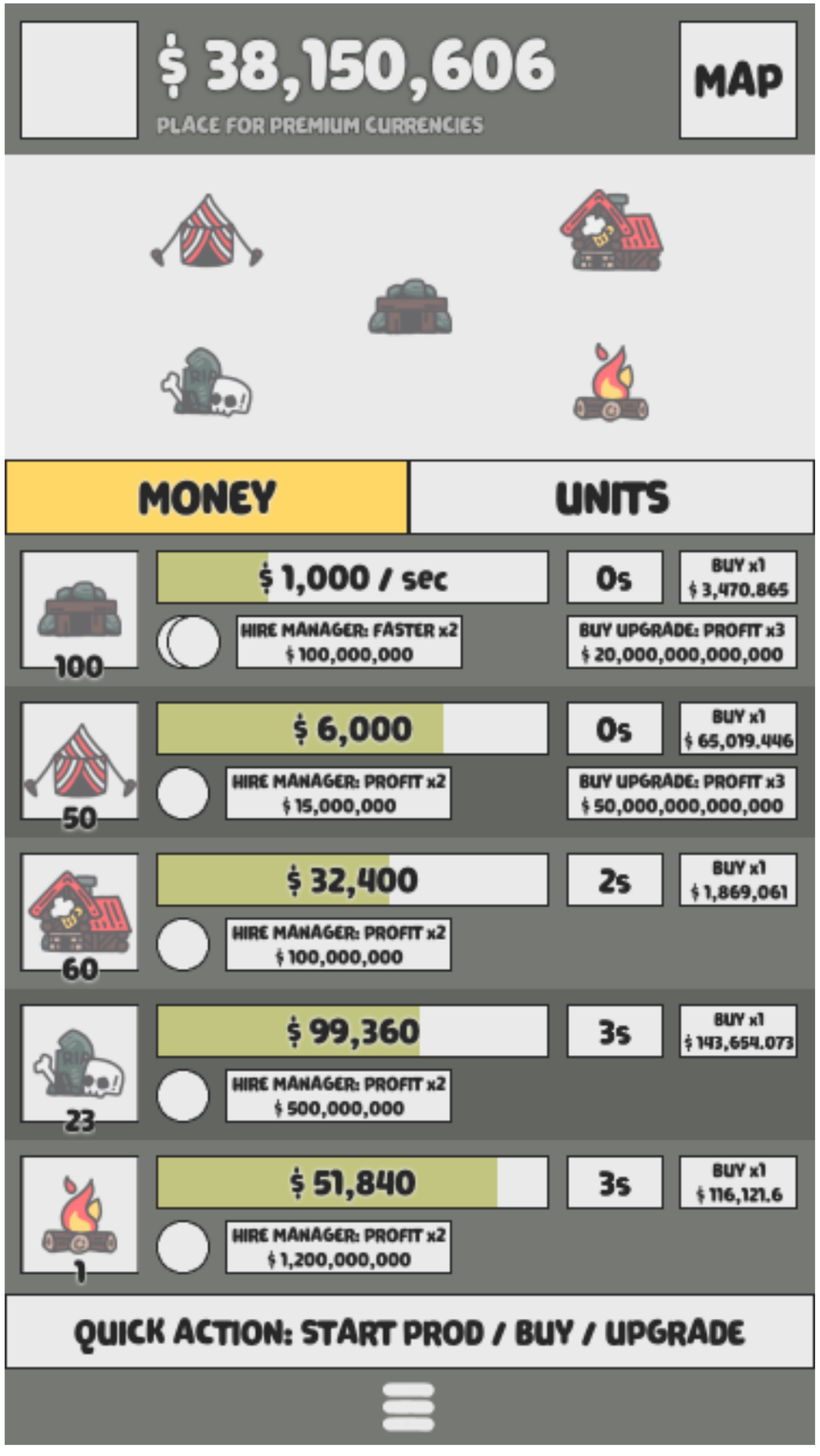
Art

Working on the game-prototype, I remembered that I'm actually not a designer or artist =) While being in the "brainstorm mode" I found a couple of asset packs which look ok-ish and I was able to utilize them in the prototype for icons and map layout.

I tried to create a simple yet fully-functioning mockup, and ended up in copying solutions from the Adventure Capitalist almost completely. Few things which I tried to add into design:

- Visualization area under header. In the current implementation it's quite simple and shows only icons of buildings, but if there were some animations and effects synchronized with generators production cycles, it would add few points for user experience and overall engaging into gameplay.
- I tried to move managers and upgrades for buildings from another screen to the production screen, to reduce amount of actions users would need to do to buy upgrades.
- The quick action button can be used not only for production, but for upgrades too.

In a nutshell: I'm not able to create a good art from the scratch, but I'm able to utilize already created design elements and combine them into some new popups/windows/components.



The End?

From the beginning I decided to write dow the time I spent for the development. And the results are quite surprising for me:

- Game Design: 9 hrs
- Art: 9.5h
- Development: Framework: 16 hrs
- Development: Server Emulator: 13 hrs
- Development: Game: 42.5 hrs
  Presentation: 8 hrs
- **Total: 98 hrs**

Wow =) That's definitely not what I expected to spend, when I started working on the prototype.

I think the main reason why I spent so much time for the Code Challenge task and the presentation is that I haven't been doing test tasks for a very long period of time and I'm actually enjoying it in some way.

I like some of the software solutions I created or improved while working on the prototype, I thought about some of them for quite some time, but hadn't had free time or energy to finish them. And now I actually have a boilerplate app which might be used for my further cross-platform pet-projects.

I would love to get any feedback from you, even if the results of my work wouldn't impress you.

Thanks!
Mark.