CTEC3451 Development Project

Splash X6 Development Report

Jordan Tyler Gray (P2540338)[1]

[1]De Montfort University

June 2, 2022

Abstract

The following is a first person account of the process and experience of developing Splash X6 from commit **eba18b2** to **1d3a733**. I discuss the research and design of the systems involved, the process of thier creation, issues I encountered, and the testing and validation involved in stabilisation.

Keywords
Game Development, LibGDX, 4X, Kotlin, Java, ANTLR, Spotify, API, Terrain Generation, Serialization.

Acknowledgements
The development of this report and it's artifacts was made possible by those who supporting the project, and myself. Firstly, in these difficult times of late, I must thank my close friend Courtney Harding alongside my close-nit support group for providing me with the mental stamina to push forward with my studies. Without their input, the toll of the times in which we live are likely to have gotten a firm grasp onto the best of me. In no particular order, I also formally thank those for their input on the project itself, no matter how trivial.

Atallah Hezbor for contribution towards the repository's configuration, and initialisation of automated cloud building using GitHub Actions, which led to the possibility of autonomous internal unit testing.

Lewis Gray who has provided artistic support & assets and, alongside Thomas Bird & Jamie Watts, has been a resource for spontaneous user testing since the original iterations of this project were proposed over three years gone by.

My university peer, and a good friend, Dylan Brand, who contributed unfathomably to the beginning of the development of the artifact. Providing conceptualisation and design input for the underlying architecture used to build the title, the audio system, and being my go-to for support when conversing through new ideas, troubleshooting, and intramural user testing; All whilst showing overall tremendous interest in the project.

Followed closely by all those who came before him in my journey of study in this field, and by no means to be considered last, my contractual product supervisor Dr. Stuart O'Connor, who takes special interest in areas of artificial intelligence which extend into the gaming domain. He has shown interest in, and support for, the concept of the project since its' proposal and has guided me along my expedition in it's realisation. His recognisably valuable opinion alone provided the validation and comfort that the project showed promise.

Contents

Source Convention

Throughout this document, the final implementation of a design mentioned will be referenced by it's location within the project's source code via it's breadcrumb through the following means :

`(impl breadcrumb.or.url.to.package.or.file)`.

A breadcrumb is typically suffix to *com.shinkson47.SplashX6*, unless it starts with *com.* Non-code file will be referenced with a directory path.

The sources to the project may be found at the URL below, which holds the latest commit at publication of this document. Note that it has been navigated to the prefix mentioned above for you.

https://github.com/jdngray77/UniX6/tree/2fbd56a146883f511e9618f3c6f0f9bd7b73db7e/core/src/com/shinkson47/SplashX6 (hyperlink)

1 Project Structure

qRegularly scheduled meetings will occur between myself and Stuart, my project supervisor, throughout. This regular constant will create structure from which organisation can be structured, where development progress and goals occur in agile-esque fortnight intervals, afterwhich, my time will be split between game and report progress. My report is not structured this way however, as I found it difficult to explain features in the correct order, in order to have both the writing and the understanding flow well.

2 Engine

My most familiar area of development is the JVM, and when it comes to Java there's only a small selection of game engines to choose from. Initially, I tried using the same foundation as Minecraft, the largest java game of all time (en fait the second largest game of all time, beat only by GTA V) (Callaghan (2016)), LWJGL but not being familliar with the low level API's involved I quickly switched to a high level engine built on LWJGL, known as LibGDX. This particular engine used the same bindings I started with, and is the only JVM engine which actively *encourages and supports* the use of Kotlin - my favorite JVM language.

3 UI / Skin

Targeting a desktop environment, I knew that the game would require a window based GUI and with the style of game in mind X6 would contain a myriad of windows. I felt it was a good place to begin.

3.1 Technical Aspects of the Skin

LibGDX's documentation showed that the interact-able front-end is implemented using the package `scene2d.ui`, a skin-able UI widget library themed through JSON metadata similar in nature to that of CSS, complimenting a draw-able texture region atlas of drawables. The use of a texture atlas offloads the UI render computation onto the GPU via OpenGL, effectively providing hardware accelerated GUI as standard.

The JSON file is loaded by `scene2d.ui`, and the visuals within are applied to the corresponding widgets. It's texture atlas is stored within a separate *.png* file, which is identified by URI in the the JSON.

These files *could* be tediously curated by hand, but referenced in LibGDX's documentation is a GUI utility called *Skin Composer*, which can develop skins graphically by importing individual visual assets and configuring presentation & behaviour of widgets. *Skin Composer* exports the data to the library requires. This tool saved masses of time and frustration in curating the skin for X6.

3.2 Design and Asset Accumulation

Not personally having the artistic ability to develop visuals I sought pre-fabricated assets in online repositories, encountering a collection of worthy candidates. Initially, X6's skin was developed following an RPG themed asset kit (Fig 1, Lamoot (2010)) which fit the nature of the game and was similar to others discovered during market research.

However, issues rapidly arose surrounding it's responsivity in implementation, particularly with the menu bar as many elements in the kit were large (Fig 7) and not fit for 10-Patch scaling, limiting the quantity of items that could be placed adjacently. The menu would overflow on my MacBook's small (1280 x 800) screen with 5 buttons. Further, a simple window would consume a large percentage of the horizontal screen real-estate. A better solution was required.

I compromised on stylisation, changing my approach to focus on usability - discarding the more appealing style and switching to a style more akin to that found in OpenTTD, closer to an old OS interface than the medieval wood. This alteration introduced 10-Patch friendly assets sourced from https://itch.io (Interactive (n.d.)), compiled again in Skin Composer. A direct contrast of the Main Menu displays how the styles differ, and how the feel was drastically altered (fig 5, fig 4).

This compromise is less visually appealing - breaking the feel of the game present with the initial assets - however in exchange X6 receives the crucial ability to fit UI entirely on-screen. For the this limited development run, I simply did not have the skill or assets to maintain the theme.

`(impl core/assets/skins/W95)`

4 Scaling

To reduce further issues of this nature, X6 would need a better responsivity solution. Being developed for desktop environments, it can assume the screen will be of a large 16:9 nature or similar.

This in mind, I could implement logic to contain a UI Stage2D stage within the screen's boundaries. The solution I designed surrounded building UI for a widescreen display, then using LibGDX's scaling functions to transform it to fit the screen.

Stage2D.ui places widgets onto a stage whose

viewport can be scaled, meaning the co-ordinate space of the Stage2D stage would match that of the render-space as determined by X6's selected display mode, but the output would be fit to the display. By default I selected the scaling mode 'Fit' to contain the stage whilst maintaining the aspect ratio, which naturally imposes a Letter Box on screens whose aspect ratios differed from the X6's selected display mode.

Now, given that the render space is not constant between workstations, widget placement must be of a relative nature inferring they are placed at the left, right, top, center, etc. as opposed to a specific co-ordinate. Elements can use 0 or the known length of the stage's axis to be placed into a corner, and I created a utility function `(impl utility.Utility#Center)` which could be used to center an object on one or both axis. Combining these extremes and centers of both axis, elements could be placed at 9 locations (Appendix 10) within the render-space.

This solution asserted objects placed on the stage would be in the location they were expected to be, and that they could not be placed off-screen.

The implementation is achieved through extending LibGDX's ScreenAdapter, creating an abstract screen with a Stage2D stage whose sized according to X6's graphical configuration.

`(impl utility.configuration.GraphicalConfig)`
`(impl rendering.ui.ScalingScreenAdapter)`

5 Reflective Options

Initially, to add some user-facing configurability into the early prototype, I added an options window and developed a 'tab' style layout (Appendix 18) where categories of options are combined into separate spaces, however this solution quickly became a developmental nuisance. Adding an option required me to script adding of a new control widget, registering Event Listener, placing a label alongside it, adding a tooltip, and ensuring it was well placed into the window.

I found scripting this kind of window to be difficult to read & maintain, and too cumbersome to quickly append an option.

To solve these issues, I'd have to automate :

- Screen placement
- Functionality of a control
- Tooltips and labels

A reflective analysis of the existing solution extracted an easily abstracted behaviour; a few kinds of controls used to change a variable. This could be abstracted with reflection if I could implement a control which would populate itself from, then modify memory to match the user's alterations, and place them into a managed layout my problems would be solved.

Desiring to use existing code, I searched through Stage2d.ui to discover it's implementation of a Tree View to be very open to extension. Using a tree would allow the window to group options by type in a hierarchy, similar to how they already in the tabs, but in a far more dynamic manor, and would eliminate manual static placements of objects and responsive issues surrounding that. The *nodes*, in such a view could be extended to create standardised controls.

I began by creating a node who could modify a variable. It accepted an object instance and the name of the field. It would also check for the presence of a *NodeInfo* annotation, a simple annotation optionally added to the variable of focus containing a tooltip which would be added to the node. This *ReflectionNode* was then extended to implement the different kinds of controls appropriate for different data types.

Lastly, I created a *RootNode* which is simply a container for children nodes. A node has this behaviour without extension, however the *RootNode* contains a constructor with far superior syntax for defining the layout of the options window, handling localisation of the category name displayed withing itself, and a vararg for it's children.

After these changes, I was no longer tasked with scripting the placement or behaviour of a control, and it's responsivity was no longer a concern, and the options' code became considerably more maintainable and readable, and options could quickly and simply be added, reducing development time wasted.

(Appendix 17)

`(impl rendering.windows.Node)`
`(impl rendering.windows.W_Options)`

5.1 Graphical Configuration

As an aside, the prior mentioned control over display modes and scaling was simply implemented using this. LibGDX retrieves a set of display configurations supported by the GPU from OpenGL the reflective options places them into a select box. A second drop-down contains the enumeration of scaling modes.

5.2 Exclusive Full-screen

A second brief aside - X6 was intended to be full-screen. Exclusive full-screen achieves marginally better GPU performance, as the soul compute time is dedicated to X6, opposed to the operating system's window environment which draws the application within itself. (Dasel (2021))

Full-screen control is implemented through my graphics configuration, held in a Kotlin object of the same name. The display mode (resolution) is stored here, and when modified is applied to the engine as either a fullscreen or windowed, according to `GraphicalConfig#fullscreen : Boolean`.

`(impl utility.configuration.GraphicalConfig)`

6 Language Localisation

Since 1997, the Java JDK has naively supported internationalisation (Oracle (1997)). X6 uses it's locale feature to implement localisation. Functionally, the behaviour of the system is similar to a map, where a key returns a value, however these values are automatically swapped with those associated with the selected language. As an example of usage, the locale may have the entry *generic.game.new*. An English dictionary may define this to be "*New Game*", whist a french dictionary may use "*Nouvelle Jeu*". With this, a button element may be assigned the key and have it parsed through the dictionary map to retrieve the correct string for the language.

X6's Localisation data was constructed English-centricly (Meaning that English is default. If no translation for a key exists, the English definition will be used.), with rough translations provided for two other languages using an online translation service as proof of concept.

`(impl core/assets/lang/lang.properties)`

`(impl utility.configuration.LanguageConfig)`

7 World Generation

Whilst analysing games in this field, I noted a feature common among them. They all have the ability to algorithmically generate their worlds. Some have prefabricated levels known as "*Scenarios*" (Dunnoob (2013)), essentially saved copies of a generated, modified, then saved world.

I wanted to implement this also since it would provide a richer experience, and further the technical portfolio of X6.

Designing a solution was initially difficult, and for some time I even believed that doing so for a tile map in LibGDX was impossible; it implements Tiled ((n.d.)) as it's 2D level implementation (LibGDX (2018)), A system designed only for loading maps hand-crafted in the Tiled editor. Luckily, whilst researching the system, I discovered that this particular implementation is partially mutable; LibGDX provides interfaces to add tiles and layers within an *existing* map. It's not possible to add resources (tilesets) however. This grants the possibility of creating a workaround whereby X6 stores a single Tiled Map (.tmx) on disk containing all tilesets that may be used, but no layers or landscape. This can then be loaded and mutated, creating tiles and layers as required to generate a map.

7.1 A Feasibility Test

After the discovery of a possible work-around, I created a test prototype to check it's feasibility which sought to implement the design brief above; Tiled was used to create and empty map containing resources, which was loaded and modified. This took place within a single file which predictably grew hard to read, update, and maintain - however my belief was proven entirely possible. The file was split into different sections of generation which were invoked in order, they itterated through the world adding tiles and layers with pseudo-random and random placements.

I felt the test script could be abstracted to a modular system where each section is a script of it's own - segregated stages which operated on a common world. Common generational utilities could be extracted into a super class to be shared. This design would confine all of the behaviour a stage to their own file and the system overall would,

as a result, become inherently modular and more maintainable. This is from where I formed the concept using a pipeline.

7.2 The Data Pipeline

The world generation system for X6 is a modular data pipeline, where generation is interpreted as a flow of world data where a collection of mutations are applied in succession, whereby the result of one mutation is the stimuli to the next. Such a design requires initial data, suggesting the pipeline must start with base data to travel through the pipeline.

This overview provides the abstraction of three types of components :

- An initial data provider, establishing the start of the pipeline. I'll call this a *Base GenerationStage*.
- A module placed after the *BaseGeneration Stage*, which mutates the result of the stage prior - A *ModifyingGenerationStage*.
- Some pipeline which encapsulates a set of stages and applies their results together as described. I'll alias this as the *Generator*, for simplicity.

Applying this all together forms the foundation of system's design (Appendices 11 & 12). To implement, I created the abstract classifications of the names above, and common mapping functions for use in stages to apply some function to every tile in the world, updating the tiles with the mutations as it does so.

Once this was in place, I began re-locating the stages from the test implementation into the hierarchy.

```
(impl game.world.generation)
(impl game.world.generation.WorldGenerationStages)
```

7.3 Land Mass

As discussed, the pipeline requires initial data. For this, I need to define some base world comprised of regions of land, and fill everything else with water.

Within the game industry, noise generation functions are commonly used to achieve this. Noise functions create pseudo-random floating-point number ranges between 0 & 1. Such noise generates randomly positioned peaks and troughs with smooth transitions between the two values and can do so within n dimensions. The predictable smoothness of this RNG makes these functions extremely useful in landscaping planes randomly. X6 will follow in the industry's steps, and use this method too.

If such a function is rendered top-down in 2 dimensions, where floats nearing one extreme are black and and towards the other are white, it can be visualised on a human level as to how it may be useful in defining pseudo random areas of land and water across a map. (Appendix 6)

In order to yield binary 'water/land' information a float, I need to draw a boundary which may be equivelated to sea level; Let my boundary be j, then every float above j can be considered land, and those below to be water. (Appendix 23)

Extending this one step further, the majority of games researched have a mechanic whereby landmass is surrounded by a shallow water, beyond which is a deep ocean which has extra requirements in order to be traversed. This can easily be implemented here by simply adding another boundary. Above j is land (grass), below j but above i is shallow water, below i is deep oceans. (Appendix 24)

In order to implement this, I would of course need a noise generation function. These are mathematically complex and difficult to design so naturally I opted for an existing implementation. A range of libraries exist for this purpose, but I settled with FastNoiseLite (Peck (2016)) - a fast and configurable library supporting a large range of noise algorithms all contained within a single portable file. I didn't enforce a particular configuration or algorithm to be used, rather I supported the use of the entire library by user-facing the options for algorithm, cell type & sizes, etc. A pleasing set of default parameters is set, making this optional to the user.

```
(impl game.world.generation.NoiseBaseStage)
```

7.4 Biomes

Biomes are regions of a map that have different generation attributes and behaviours, which can be used to generate different environments around the map in order to provide some variation to the land mass. As an example, some areas may be grassy plains, whilst others may be a sandy desert. Later on, these biomes could provide different resources

and have different requirements, leading to them having an impact on the game-play.

This is commonly achieved with the use of a voronoi diagram. Such a map chooses n points on the plane, placing a circular cell at each point and expanding it's radius until it meets another cell, and cannot expand any further. The resulting pattern is reminiscent to that of a giraffe's spots, or balloons being inflated simultaneously in a box. Each cell can then be randomly assigned a biome, and the whole pattern overlaid into the world to modify the existing landmass.

I used a short voronoi generation script from rosettacode (Smarmius (2011)) `(impl com.rosettacode.voronoi.Voroni)`. It was created to draw the cells using AWT, so I modified it under license to evaluate the cells as members of my Biome enumeration `(impl game.world.Biome)`, which is a list of all terrain types in the tileset. A modification stage was added to the pipeline to use this diagram to mutate the world, changing the grass landscape into the other materials such as sand or snow.

`(impl game.world.generation.stages.VoronoiBiomeModStage)`

7.5 Height Map

To make the flat landscapes more interesting and adding further regions which require researched skills to traverse, another modification stage was created which adds ranges of hills and mountains to the landscape. This was achieved in a fashion very similar to that of the base stage, where noise was used alongside two configurable boundaries - one for hills, the other for mountains. Where in the land generation the noise would generate land surrounded by shallow water, the same logic would generate mountains surrounded by hills to act as a similar gradient shift as the height map transitioned from trough to peak.

Here, however, a differently seeded and scaled noise generator is used, drawing smaller noise regions such that many small peak regions would fall within a single biome. The output of this generator in context to this stage is known as the *height map*.

This stage is limited by the tilesets, which only contain hills and mountains for select landscapes, leading me to add a filter for these *heightable regions* to the stage so it would only attempt to apply the height map to these biomes. `(impl game.world.generation.stages.HeightModStage)`

7.6 Foliage

Further decorating the landscape is some plantation, generation of which is entirely random in placement. The stage chooses n random places on the map, and compares them against a map which simply determines what plantation to populate a biome with.

7.7 Interpolation

At this stage of the pipeline, the structure of the world is complete. All that remains is some miscellaneous steps.

First of which is tile interpolation; the boundary blending between different types of tile. This applies to land, water, and height tiles. The tileset contains all possible combinations of blended tiles, X6 just needs to analyse each tile and it's neighbours to determine if it should be replaced with a slightly different tile which blends itself with it's surroundings.

The tileset maps tiles by names in the convention of $x\_x\_x\_x$, where each x is the resource name of a tile on one side, in the format $north\_east\_south\_west$. The interpolation stage needs to simply generate a string in this format for every tile, filling in the gaps with the resource name of the neighbors. If the tile is at the edge of a map and has no neighbor, then the gap is filled with the name of the tile being analysed, thus that the side is unaltered.

For simplicity of explanation, I'll use verbose titles of the tiles as opposed to resource names. Suppose the interpolation stage is analysing a grass tile of which to the north and east is grass, to the south and west is water. The tile being analysed will be replaced by the tile retrieved by the query $grass\_grass\_water\_water$; a predominantly grass tile which fades to water along two sides.

The effect of this feature can be observed in between Appendices 21 & 22.

Almost identical logic is applied to height tiles, raising or lowering sides of hills and mountains to meet the surrounding terrain. `(impl game.world.generation.stages.InterpolationModS`

`(impl game.world.Tile#interpolate)`

7.8 Navigation Data Layer

The final layer is the navigation layer which contains a pathfinding graph which is not visible to the user. Each cell in this layer contains a boolean as to weather it can be traversed or not by land units.

In game, when it comes time to find a path through the world for a unit's travel, the algorithm used will repeatedly ask the game "*Can I go here @(x,y)?*", to which X6 will examine the matching cell in this layer and respond with the boolean found in the cell. This feature is described further in 11

This implementation limited the game, since all units share the same land based data they cannot have unique travel abilities; i.e a boat will travel only on land in the same fashion as a land unit, and all land units will avoid mountains. Future revisions will need to see though adding more layers for different categories of units.

7.9 Layers

Tiled maps may contain multiple overlaid planes, permitting a tile to be above another, and different kinds of world data to be stored independently. Segregation allows for polling presence of objects by type within a given cell by simply comparing the data at the a co-ordinate in the correct layer against null. X6 will use layers to place features above the land, most notably the foliage and hills. These layers can also contain non-tile data, X6 uses a *NavigationTiledMapLayer* introduced by a library mentioned in path-finding (11) to store a navigational data graph for moving units.

(Appendix 27)

8 Fog of War

A common mechanic whereby the player may only actively observe the immediate area surrounding characters they own. Some implementations, such as that in FreeCiv, also display previously observed areas of the map in the state of which they were observed.

I developed X6's fog of war prototype in a similar fashion, where the map would be revealed as the played explored. I later discovered that the design of the world and the game's engine made displaying areas of the world without updating them to reflect the current state to be impossible, alas, another compromise; I introduced lighting onto characters, darkening the entire map except for a spotlight around units (Section 12), creating a line of sight style obfuscation which results in areas of the map being unrevealed unless previously charted, and only tiles within an radius surrounding a unit to be observable. (Appendices 19 & 20)

The square radius of defog is data collected from FreeCiv rulesets for classifications of units, and their abilities. (FreeCiv (1999*d*))

9 FreeCiv Data

The game required data for a few systems. The world, as discussed, requires various tilesets, entities need sprites, and research trees and dependency rules will be needed throughout to manage the players ability to perform things during the game.

Initially, I looked for tilesets on https://itch.io, however I ended up setting with harvesting assets from the FreeCiv project; being open source in nature, the assets are freely available for viewing. The project's GPL-2 license permits its use, modification and distribution.

Reaping from this repository provided a consistent source for a range of assets that would be required, in visually the same visual style. Sprites were compiled into a spritesheet using TexturePacker. (FreeCiv (1999*d*), FreeCiv (1999*a*), FreeCiv (1999*b*), FreeCiv (1999*b*))

10 Turn Hooking

From market influence, X6 is a turn based game. Whilst a lesser implementation may contain a large method to executed every turn, I found that whilst scripting there was commonly a desire to receive notification into a feature of some kind when the user ended a turn, ranging from evaluation of production systems to updating of state machines and of windows reflect the state of the game.

My solution to this desire was to introduce an interface similar to that of a web hook, where some script desiring turn-over notifications would implement this *hook* interface and subscribe it into the game's main controller (the *GameHypervisor*). The *GameHook* interface is simply an alias subtype to JDK's Runnable interface, where it

renames renames the `run` method to `onTurn`. `(impl utility.TurnHook)`

The Hypervisor would store a list of subscribed *GameHook*'s, and invoke them all on a turn's end.

11 Pathfinding

As discussed in World Generation (7.8), a world layer contains a path finding graph. In order for units to utilise it, X6 would need a path finding algorithm. Once again, I opted for an existing implementation for simplicity in implementation, avoiding the developmental complexities and inevitable issues creating my own would bring.

I settled with a LibGDX supported implementation of the AStar path-finding (Guzman (2014)), a generally flex-able and easy to use algorithm. The library introduces the *NavigationTiledMapLayer* mentioned earlier, containing a graph of tiles that may and may not be travelled. Each unit contains an `AStarGridFinder`, a walker which navigates the graph to determine the shortest path from the units present location to it's desired destination.

The finder returns a list of co-ordinates determining a path which may be taken, which the unit can then traverse on it's turn hook. It does so by dropping $x-1$ co-ordinates, determined by the unit's travel distance in the FreeCiv data, and teleporting to the new head of the list. The finder returns null if it is not possible to path to the desired location, in which case the unit clears it's destination.

12 Nations & Units

A game contains many players, each with a monopoly over a Nation and it's resources. The imported FreeCiv data surrounding nations contains information about thier leaders (Unused), the type of sprites to use in settlements, and a roughly chronological ordered list of city names.

A nation may have a collection of tactical entities, known as 'Units' - the Sprites for which are also sourced from FreeCiv, as is the enumeration of classifications of unit.

The unit implementation extends the on-screen sprite. The unit's classification is used to determine which sprite is used to represent it and what actions are retrieved from the *UnitActionDictionary*, determining what actions the unit is able to perform in-game.

12.1 Unit Actions

When a Nation ends their turn, it's units may perform some kind of action. For example, a unit may travel around the map towards their destination, attack another unit, or special abilities provided to them based on their class - i.e a only a settler is able to settle a city. To implement a dynamic action I chose to use Runnable scripts as apposed to determining behaviour by branching code paths. This choice reveals a more dynamic design frame whereby a unit holds a `Runnable turnAction` which can be invoked by a `TurnHook`. Also held by the unit is a list of actions available to the unit, such that a user can choose which action to run from those available.

Since the actions are Runnable instances, they are inherently objects the JVM memory heap. If there were unique instances for every unit, the heap would quickly be overwhelmed by duplicate instances of objects which perform the same behavioural script. Instead, if the actions were written like a `Consumer` where they receive a reference to the unit on they are to operate, then the application could hold a single instance that could operate on any given unit. This memory saving is the secondary role of the *UnitActionDictionary*, implemented by defining the actions as final objects. Primarily however, the dictionary determines which actions are available to a unit by class, which is achieved by extending a Map, being `Map` of `Class -> { actions* }`. Some actions are available to all units, so I inserted a pseudo-class into the enumeration titled `_BASE` to which these common actions are assigned, and thus when the dictionary is accessed the response set is union to `_BASE` and the classification. (`UnitActionDictionary#get(UnitClass)= \_BASE` $\cup$ `C`)

`(impl game.units.UnitActionDictionary)`

13 Settlements

Instantiated into a nation by settler class units, Settlements are a game feature whereby a nation can create hubs of production throughout the world. Settlements contain production systems for creating new units, alongside improvements to the settlements. Similar to units, settlements also expends a sprite - determined, this time, by the nation data as mentioned prior.

14 Production

Abstractly, a production system in X6 is method of creating or completing something over the course the game. Being a production hub, a settlement produces both units and improvements in paralell, in conjuntion to the nation itself which can complete research projects to make available greater units. The production system, needing to be re-used this way, will need to be implemented just as abstractly as described.

To start, a **ProductionManager** to oversee a production system, whose job is to queue and complete **ProductionProjects** in turn. A project has an associated cost, and the manager has some quantity of resources to apply towards that cost every turn.

Take Appendix 5, where there are three projects queued, with no contribution applied. Every turn, the manager will iterate through it's queue, applying it's production power such that is the quantity available is consumed as it does so. From another perspective, the manager applies it's power to the first project, and the remainder to the next, repeating until the available resource has been consumed. This can be described in pseudo :

```
val power = x
var index = 0
while (true) {
    power = project[index].contribute(power)

    if (power == 0) break
    else index++
}
```

As an example, take listing 5, where there are three projects in the queue. If a manager with 30 power was executed on this queue the first project would be completed and the second would have 5 cost remaining. The result of this operation is displayed in listing 6.

The class contains abstract methods for the implementations to populate the differing information, particularly the **EvaluateProducable** which generates a list of **ProductionProject**'s which may be generated at present for the manager to have available, the cost of the **ProductionProject**, and the **doClaim** which claims the result of a project once it has been completed.

A standardised UI pane displys the state of a manager (appendix 14) and allows the player to manupulate it's queue. As mentioned, three of these are implemented for Unit, Improvement and Research production.

```
(impl game.production.ProductionManager)
(impl game.production.ProductionProject)
```

15 Research Tree

The player is limited by their research, as they complete research projects more units and city improvements become available to them.

Data for techs were, like the other data, reaped from FreeCiv (FreeCiv (1999*c*)). I constructed a tree data structure that would build from the dependencies within the technology data. Level 0 in the tree required no other techs, Level two depended on something from level one, etc. Each consecutive level was represented to the user by placing them in columns (Appendix 26). I organised the production system uniquely such that it would, for a given selected tech, calculate all of the dependencies and place them into the production queue in order of dependency levels (Appendix 25).

The unit data used in 12 held tech dependencies so I could, with relative ease, modify the unit production's **evaluateProducable** to make only those with no or completed dependencies available to be produced.

16 Serialization

The game data will need to be portable for two purposes :

- Saving a game locally to disk

- Sending the state of a game over the network for multiplayer

The best approach to a custom multiplayer architecture would be a client-server relationship with a solid custom protocol, similar to that of the Minecraft protocol (Sadimusi (2011)). However, with the time constraints given, the client side logic for loading a serialized game can be re-used for networking also - making it possible for clients to share a game over the network.

The game data *was* stored in a kotlin object under the same name, however to make it serializable it must be a class instance that is a subclass of `Serializable` so the object was mutated to a class with the name **_GameData**, and the common instance was kept under a variable with the origional name **GameData**. This container cascades to many other classes and data, many of which are not serializable - posing an issue. Non-serializable members can be excluded from the serialization with the `\@Transient` annotation, however on a case-by-case basis they must be re-constructed when loaded.

For this, I created a **PartiallySerializable** interface - which extends **Serializable** simply to sport a virtual **deserialize** method. Upon load, deserialize is invoked to re-construct or initalise these transient components.

As an example, the Tiled world layers, containing the terrain as renderable sprites, are transient but required by client to render. The world data itself, however, is stored separately as required by the fog of war system, & tiles are only added to the layers when a area is defogged. On deserialize, the world data is transposed into the layers using the de-fog logic. Since, for now, X6 does not store a log of which areas have and have not been defogged, the client must reveal the entire world - regardless of exploration.

(`impl com.shinkson47.SplashX6.utility.PartiallySerializable`)

16.1 Lan Multiplayer

In order to then share this directly over the network, I developed a rough client-server implementation. The hosting game acts as a server, sharing it's game data to the other clients.

Whilst creating a game, the server accepts incomming connections as players joining the game. The clients are identified by a username the user provided when prompted, and are assigned a nation. After the game has started, these players must already own a nation in order to join, else the connection will be refused. To allow one player to act at a time, the GameData holds an integer representing the index of player whose turn it is. When sent to all clients, this index is compared against the local player to see if it the client's turn, and the **GameScreen**'s UI stage is hidden if it's not.

When a player ends thier turn, that client's game state is returned to the server, who shares it back to all clients who automatically hide/show thier UI and notify the player as to whose turn it is. AI nations are invoked on the host client, and they end thier turn immediately after invoking the state machine.

The host style architecture of this design was a mistake, it was not thought out. Clients have direct control over manipulating the game state; the server cannot retain control and assert that only the correct player takes a turn if the open-source client was modified. Further, not all clients are treated equal - which is becomming a large issue. The host client is concidered the 'master' which **is** the server, and the other's as 'slave's. The two classifications have entirely different responsibilities and interfaces to the game.

17 State Machine

A Fininte-State Machine (FSM) is a logical behavioural system, whereby the exhibited behaviour is determined by current state. One functional requirement for X6 was that of opponents, and adding AI players would permit single player game-play.

Collating previous experience of working with such machines in Unity, I set about creating an implementation for the game. I define a state machine to contain a collection of possible states, where one state is active at any time. The machine switches between states when a predicate defining a transition from the present state to another is met. Each state contains three types of behaviours. The main behaviour is exhibited when the machine is updated (i.e upon ending turn, or every frame; dependant on use of a given state machine.), two other behaviours execute when that given state is entered or exited.

Fig 8 models a pseudo state-flow to demonstrate this.

This definition was implemented with an object oriented approach, classifying the State with three Runnable behaviours, and a map of `Predicate => State` defining states which this one could lead to, and when to change to them.

17.1 State Machine Scripture

My implementation's API was streamlined & structured logically (Listing 2), however I found it repetitive, tedious, and generally not human-friendly to design and write state machines in this way. I focused on how this could become a significant issue when X6 begins using state machines more prevalent; Recognising the repetition and structure of the API would be easy to have automatically generated, and ventured to design and implement a human-friendly script to define machines that would compile into the exact same Java code. I also have some previous knowledge and experience with using ANTLR (Another Tool for Language Recognition) to create custom languages, and elementary virtual machines to execute them, leading me to believe that I was perfectly capable of solving this issue.

Design began by writing a pseudo script in a format that I found intuitive, encompassing all properties that my State Machine implementation needed. Listing 3 shows the result. The result was far easier to develop, read and maintain than listing 2, so I went about defining the grammar this format in an ANTLR G4 file (Listing 4). Java code blocks are accepted in State Machine Scripture - A custom scripting language for state machines. (SMS) where they would constitute the actual exhibited behaviour of the states; however, to avoid defining Java's grammer in order to achieve this, I opted to import an existing G4 covering Java's syntax.

ANTLR then generated Java-written tokenisers and traversers for my new language. These files made it possible for me to write a compiler for the SMS compiler in less than 300 lines, in order to translate the SMS into Java.

(`impl statescript/src/com/shinkson47/SplashX6/ai/statescript/grammar/StateScript.g4`)

(`impl statescript/src/com/shinkson47/SplashX6/ai/statescript/SSVM.kt`)

17.2 The Compiler's Result

This human-friendly SMS code :

```
state Travel {
    behaviour {
        // Any valid java code.
        int exampleCode = 0;
        exampleCode++;
        return;
    }
}
```

Is translated by my compiler into the following Java code for use in the game :

```
addState(new State(
        // State name
        "Travel",

        // Main behaviour
        () -> {
            // Any valid java code.
            int exampleCode = 0;
            exampleCode++;
            return;
        },


        this, // Parent state machine ref
        null, // Enter behaviour
        null  // Exit behaviour
    ));
```

18 Spotify

An optional extra to satiate an avid Spotify listener with a desire to use an API, X6 contains a window to manipulate an active spotify session.

X6 uses a Java wrapper to carry the heavy work of communicating with the web-API and interpreting its JSON, providing the game a comparatively simple object oriented interface. I wrap this further, extracting the most fundamental operations into functions for the game to use, i.e `Spotify#pause`, which will then execute the appropriate POST request asynchronously.

The most complex part of this system is the authentication (Appendix 13), which consists of 6 major steps :

- Request permission to perform specific requests

with a user's account

- User logs in, reads the request and consents
- Spotify provides a temporary authentication token.
- Using this temporary token, the application requests access and refresh tokens, which are stored.
- Requests are made with the access token, whilst valid.
- Once expired, the refresh token is used to request a new access token.

The wrapper navigates to a URL in a local browser to complete the first two steps, and spotify sends the authentication to a listening API - this poses a problem. The API is designed for web applications with domains for Spotify to send this to. Luckily, I own a personal domain - and whilst I cannot use it to host a listening web app that is somehow connected to the game, I can create a page on my website to accept the authentication token as a HTML parameter and display it to the user, who copies this key, returns to X6, and dismiss the instructional dialog. The key is automatically read from the clipboard, and the other steps follow.

The end result of this is an in-game window able to manipulate an active spotify session on the users account. (Visible in Appendix 3)

19 User Testing

In order to determine of the product was viable, and to determine issues, I chose to do a user-facing examination. To achieve this, I made a Google Form which reaped and categorised improvement oriented feedback; accepted criterium were :

- Crash
- Confusion
- Bug (Unexpected behaviour)
- Other improvement suggestion

35 responses collected around period 6 from peers, friends & family provided a range of data. Period five's progress meeting remarked improvements in stability were desperately required. The trial occoured *after* logging, triaging and solving issues (Specifcally predominantly commit 08f9936. Appendix 16). However, my mindset at the time was focused on the instability - placing the product in the hands of many new users lead to my prediction that many new stability issues would surface.

Contrary to my expectation, raw result data showed the major issue category was in-fact with confusion (Appendix 15). There were still some crashes documented, however the predominant distribution being **42.9%** clarity oriented.

Following up with some of the participants, I found a common factor was a lack of experience with this catergory of strategy game leading to a soul reliance on the limited in-game help text.

20 Was It A Success?

The game has implemented a major range of features from different categories of development. Extensive unit testing was not carried out within the time frame however based on my own usage and observations of others interactions with the product, I hypothesise it would show that features were implemented to a generally acceptable standard for an alpha state game, meeting the outlying objective of the contract. In direct contradiction, usage of the game also reveals various usability issues and crashes, indicating that the project requires more time to perfect the implementations to obtain practical elegance in execution.

With the audience who test drove the client, it's difficult to definitively answer this question. The data has lesser value since it almost entirely reveals confusion from those who have little interest / experience with strategy games, as opposed to *this* game in comparison to others of a similar nature. I believe that if the game were examined by players of the inspiring games, or more generally strategy games, the results may show a different story, and hold more value in representing the objective success of the project.

Overall I believe that the feature goals set out were achieved to an alpha level, the features *were implemented* but require more polish. Game-play as a whole is not yet complete, and as for play-ability in the face of end-users I cannot draw a definitive conclusion with the data achieved from this population.

21 The Future

The foundation of this prototype fulfills the goals I set out for. To date this is my largest personal developmental project, a journey along which I learnt a range of new skills, and pushed my abilities to implement features for the first time. Non-exhaustively the project, so far, has led me through :

- Learning a new, modern, programming language.
- Learning and using a new game engine with true hardware acceleration
- Creating my own scripting language using ANTLR, and a compiler for it.
- Using a web API for the first time
- Implementing my own finite state machine system
- Creating a modular world generation system
- Examining open source repositories of similar games to observe how others implemented these features.
- Data reaping and formatting from third parties
- Handling external assets under a myriad of Open Source licenses.

Looking at back at what the project *has* achieved, and the large collection stable non-visible programmatic utilities that is has resulted in, I'm overjoyed with the progress of development. Alas, the project is not complete - there's much improvement to be made according to my own usage and observation, alongside user feedback.

There are also many improvements that I would like to see through, including the networking architecture. Multiplayer is demonstratably functional as a proof to the possibility, however the architecture is poorly designed. Since the hosting client directly acts as the server, this inherently means that not all clients are treated equal. There is a lack of a keep alive feature, graceful disconnect handling, general standardisation of communication, and within the code the lines of data blurs between local and global (eg. the global **GameData** has the member **selectedUnit** - a reference to a unit to which the local client is operating. Yet it is shared amongst clients). Once this deliverable has concluded, this architecture will need an overhaul such that the game persists server side, and every client is treated equal with a *request - response* design, even in single player.

Furhter, A UI utility toolbox I had created was never tested independently, and is now filled with messy, inconsistent, hot-fixed methods, and yet it is the foundation of every in-game window. It creates a lot of development quirks with creating the UI, providing some ease of development but also creating a larger learning curve than necessary in contribution. There are also un-solveable engine-side issues with render resolution and exclusive full-screen switching on some platforms - however since there's no maturer, feature rich engine as LibGDX for this platform, it's unlikely X6 will ever be able to break free from some of these blatant issues.

My final remark -
the prototype is functional and it's development has furthered my scope of skill-set greatly, however it's clear that there's still a much work to be completed on the game in order to both mature its stability, and overhaul it's server-client architecture, before any further extensions can be completed.

References

(n.d.).
URL: *https://www.mapeditor.org/*

Callaghan, M. L. (2016), 'Minecraft is now the second most popular game ever'.
URL: *https://www.popsci.com/minecraft-sells-over-100-million-copies/*

Dasel (2021), 'Fullscreen options: Exclusive fullscreen vs. fullscreen window (borderless)'.
URL: *https://support.unity.com/hc/en-us/articles/115001276723-Fullscreen-options-Exclusive-Fullscreen-vs-*

Dunnoob (2013), 'Scenarios'.
URL: *https://freeciv.fandom.com/wiki/Scenarios*

FreeCiv (1999*a*), 'Freeciv city data'.
URL: *https://github.com/freeciv/freeciv/blob/master/data/civ1/cities.ruleset*

FreeCiv (1999*b*), 'Freeciv nation data'.
URL: *https://github.com/freeciv/freeciv/blob/master/data/civ1/nations.ruleset*

FreeCiv (1999*c*), 'Freeciv tech data'.
URL: *https://github.com/freeciv/freeciv/blob/master/data/civ1/techs.ruleset*

FreeCiv (1999*d*), 'Freeciv unit data'.
URL: *https://github.com/freeciv/freeciv/blob/master/data/civ1/units.ruleset*

Guzman, X. (2014), 'Astar pathfinding'.
URL: *https://github.com/xaguzman/pathfinding/blob/master/pathfinding/src/main/org/xguzm/pathfinding/j*

Interactive, C.-. (n.d.), 'Retro windows gui'.
URL: *https://comp3interactive.itch.io/retro-windows-gui*

Lamoot (2010), 'Rpg gui construction kit v1.0'.
URL: *https://opengameart.org/content/rpg-gui-construction-kit-v10*

LibGDX (2018), 'Tiledmap (libgdx api)'.
URL: *https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/maps/tiled/TiledMap.html*

Oracle (1997), 'Locale (java internationalisation)'.
URL: *https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html*

Peck, J. (2016), 'Fast noise lite'.
URL: *https://github.com/Auburn/FastNoiseLite*

Sadimusi (2011), 'Minecraft protocol'.
URL: *https://wiki.vg/Protocol*

Smarmius (2011), 'Voronoi diagram - rosetta code'.
URL: *https://rosettacode.org/wiki/Voronoi*$_d$*iagramJava*

Wikimedia Commons, B. a. E. (n.d.), *English: 2D Perlin noise made by Burgercat in Processing.*
URL: *https://commons.wikimedia.org/wiki/File:2D*$_s$*ample*$_o$*f*$_P$*erlin*$_n$*oise.png*

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.4" tiledversion="1.4.3" orientation="staggered"
    renderorder="right-up" width="1" height="1" tilewidth="64"
    tileheight="32" infinite="0" staggeraxis="y"
    staggerindex="odd" nextlayerid="2" nextobjectid="1"
>
 <tileset firstgid="1" source="tsx/t.tsx"/>
 <tileset firstgid="217" source="tsx/s.tsx"/>
 <tileset firstgid="433" source="tsx/p.tsx"/>
 <tileset firstgid="649" source="tsx/g.tsx"/>
 <tileset firstgid="865" source="tsx/d.tsx"/>
 <tileset firstgid="1081" source="tsx/a.tsx"/>
 <tileset firstgid="1297" source="tsx/foliage.tsx"/>
 <tileset firstgid="1376" source="tsx/oceans.tsx"/>
 <tileset firstgid="1392" source="tsx/hills.tsx"/>
 <tileset firstgid="1408" source="tsx/mountains.tsx"/>
 <layer id="1" name="Tile Layer 1" width="1" height="1">
  <data encoding="csv">
1393
</data>
 </layer>
</map>
```

Listing 1: An empty tiled map (.tmx), containing references to all tilesets (.tsx).

```
...
super("WorldCreationScreenController");

// Signature :
// addState(new State(Name: String, behaviour: Runnable?, ownedBy: StateMachine, enterScript: Runn

addState(
    new State(
        "GameConfigure",
        null,
        this,
        () -> {
            addw(gameCreationWindow);
            Gdx.input.inputProcessor=stage;
        },
        () -> {
            if(isConnecting) {
                constructConnectingText();
            } else {
                constructLoadingText();
            }
        }
    )
);

// Switch : from GameConfigure to PreRender

// Signature :
// registerSwitchCondition(fromStateIndex: Int, toStateIndex: Int, when: Predicate<Unit>)

registerSwitchCondition(0, 2, unit -> Client.DEBUG_MODE | isConnecting);

...
```

Listing 2: An example of the API methods used to construct a state machine in X6's State Machine implementation. It's a snippet of the compiled version of 3

```
/**

The state in which the user is presented
a window, and configures the game's settings.

*/
state GameConfigure {
    enter {
        addw(gameCreationWindow);
        Gdx.input.inputProcessor = stage;
    }

    exit {
        if (isConnecting ) {
            constructConnectingText();
        } else if (isDeserializing) {
            constructDeserializingText()
        } else {
            constructGeneratingText();
        }
    }
}

switch from GameConfigure to PreRender if (Client.DEBUG_MODE | isConnecting);
```

Listing 3: A snippet of X6's custom state machine scripture defining part of the loading screen behaviour. Effectively compiles into Listing 2.

```
grammar StateScript;
import Java;

// KEYWORDS
STATE      : 'state';
BEHAVIOUR  : 'behaviour';
ENTER      : 'enter';
EXIT       : 'exit';
CODE       : 'code';

NAME       : 'name';
DESCRIPTION : 'description';

FROM       : 'from';
TO         : 'to';

// ignore whitespace & comments.
WS : (' ' | '\t' | '\n')+ -> skip;
CMT: COMMENT -> channel(HIDDEN);

name: NAME ASSIGN Identifier SEMI;
description: DESCRIPTION ASSIGN StringLiteral SEMI;

defaultState: DEFAULT ASSIGN Identifier SEMI;

behaviour  : BEHAVIOUR block;
enterScript : ENTER block;
exitScript : EXIT block;

code:
    CODE
    classBody
    ;

switchState: SWITCH FROM Identifier TO Identifier (IF parExpression)? SEMI;

state: STATE Identifier
    LBRACE
    behaviour?
    enterScript?
    exitScript?
    RBRACE;

script:
    name
    description?
    defaultState

    (importDeclaration|code|state|switchState)*;
```

Listing 4: The ANTLR (Another Tool for Language Recognition) grammar file which defines the custom State Machine Scripture - A custom scripting language for state machines. (SMS)

```
+----------+--------+-----------+--------+
|Project   | Cost   |Contributed|Complete|
|          |        |           |        |
|Project 1 | 20     |0          |        |
|          |        |           |        |
|Project 2 | 15     |0          |        |
|          |        |           |        |
|Project 3 | 10     |0          |        |
+----------+--------+-----------+--------+
```

Listing 5: Three queued production projects.

```
+----------+--------+-----------+--------+
|Project   | Cost   |Contributed|Complete|
|          |        |           |        |
|Project 1 | 20     |20         |   X    |
|          |        |           |        |
|Project 2 | 15     |10         |        |
|          |        |           |        |
|Project 3 | 10     |0          |        |
+----------+--------+-----------+--------+
```

Listing 6: 5 after applying 30 contribution power.

Figure 1: Preview image of the retro RPG asset kit in use.

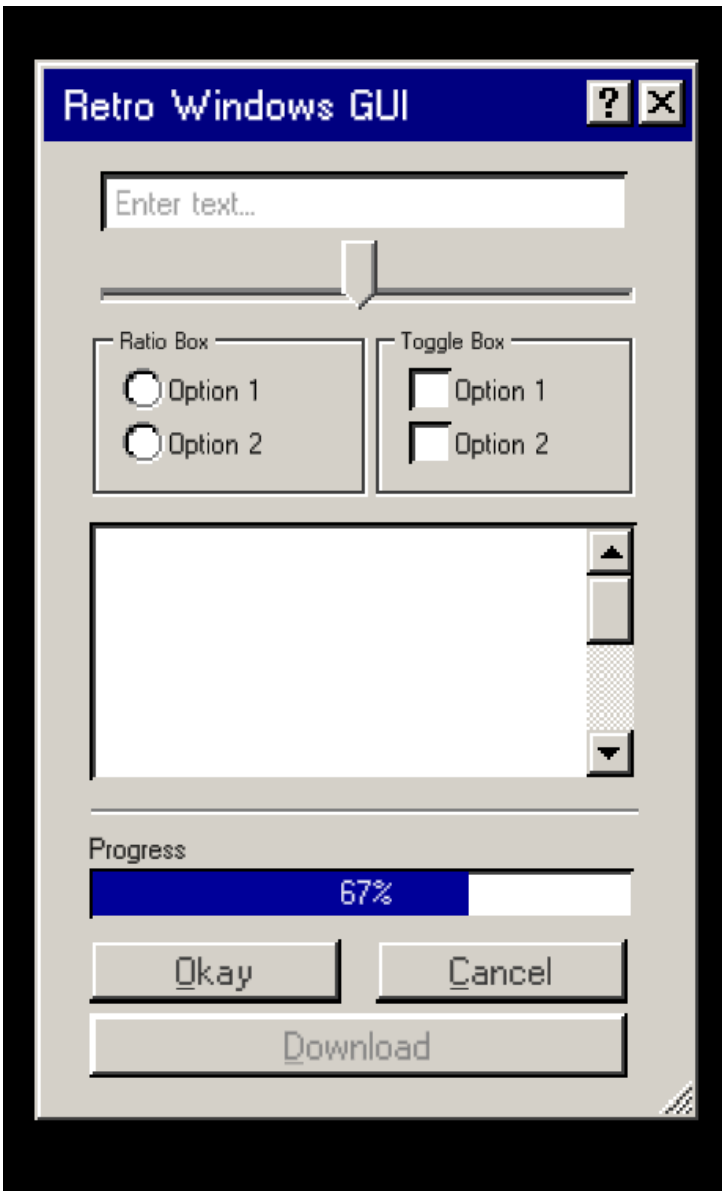

Figure 2: Preview image of the retro GUI asset kit in use.

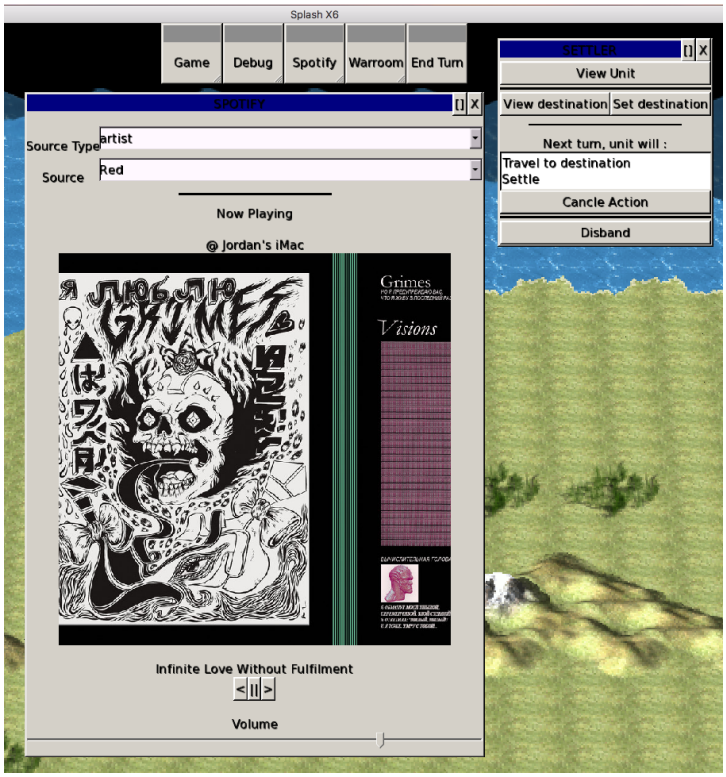

Figure 3: Screenshot of game elements with the new retro theme.

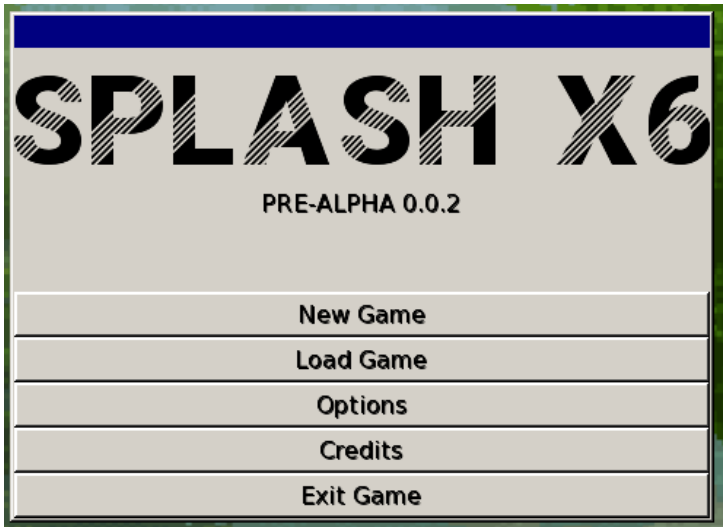

Figure 4: Screenshot of the main menu window with the new retro theme.

Figure 5: Screenshot of the main menu window with the rpg theme.

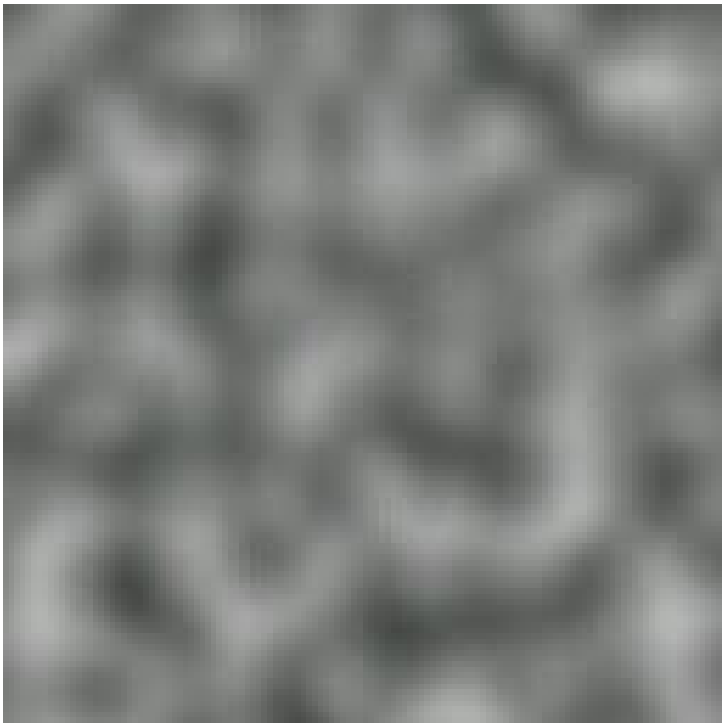

Figure 6: (Wikimedia Commons (n.d.)) A rendering of perlin noise.

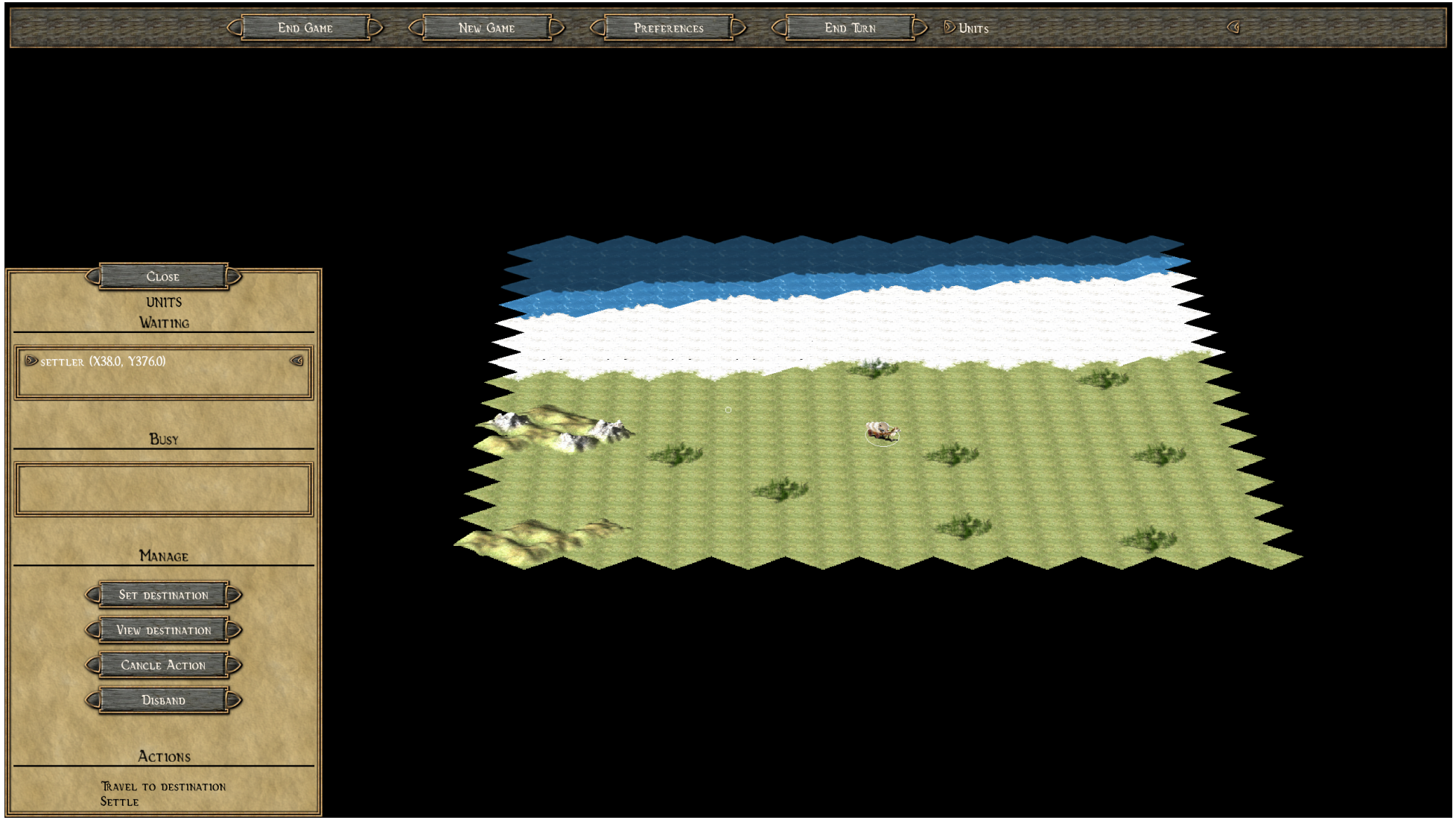

Figure 7: Screenshot of the game screen with the rpg theme.

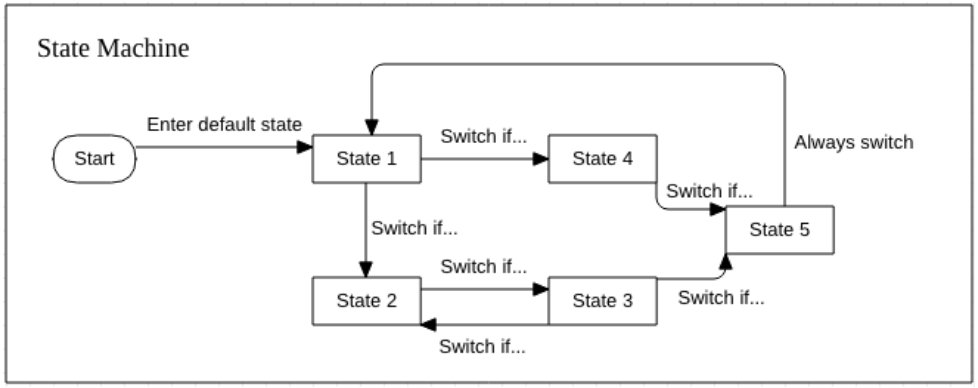

Figure 8: Pseudo diagram visualising what a state machine's flow between states could look like.

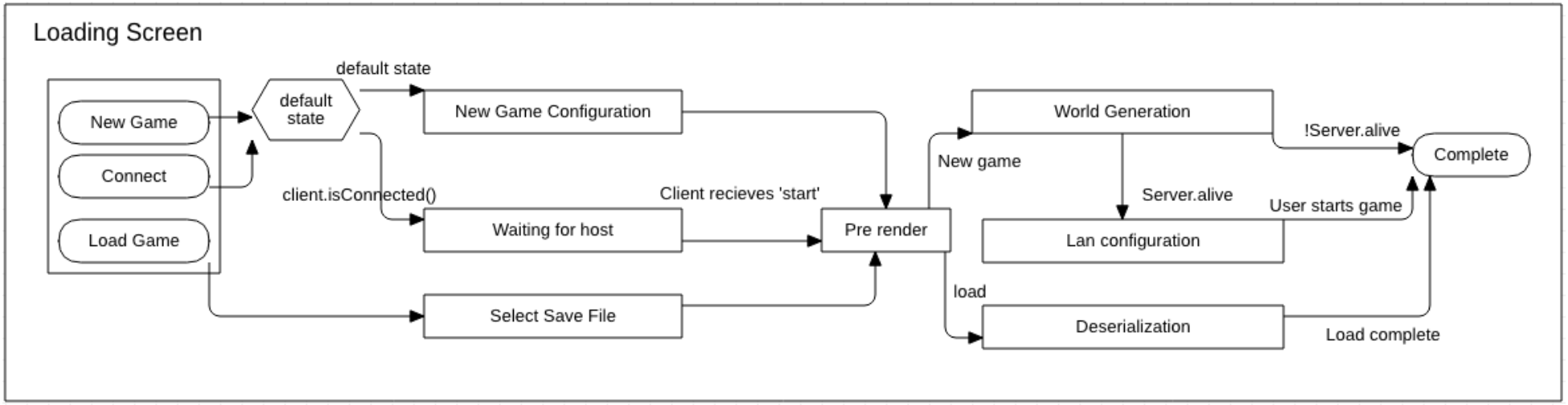

Figure 9: The flow of states in the loading screen's state machine.

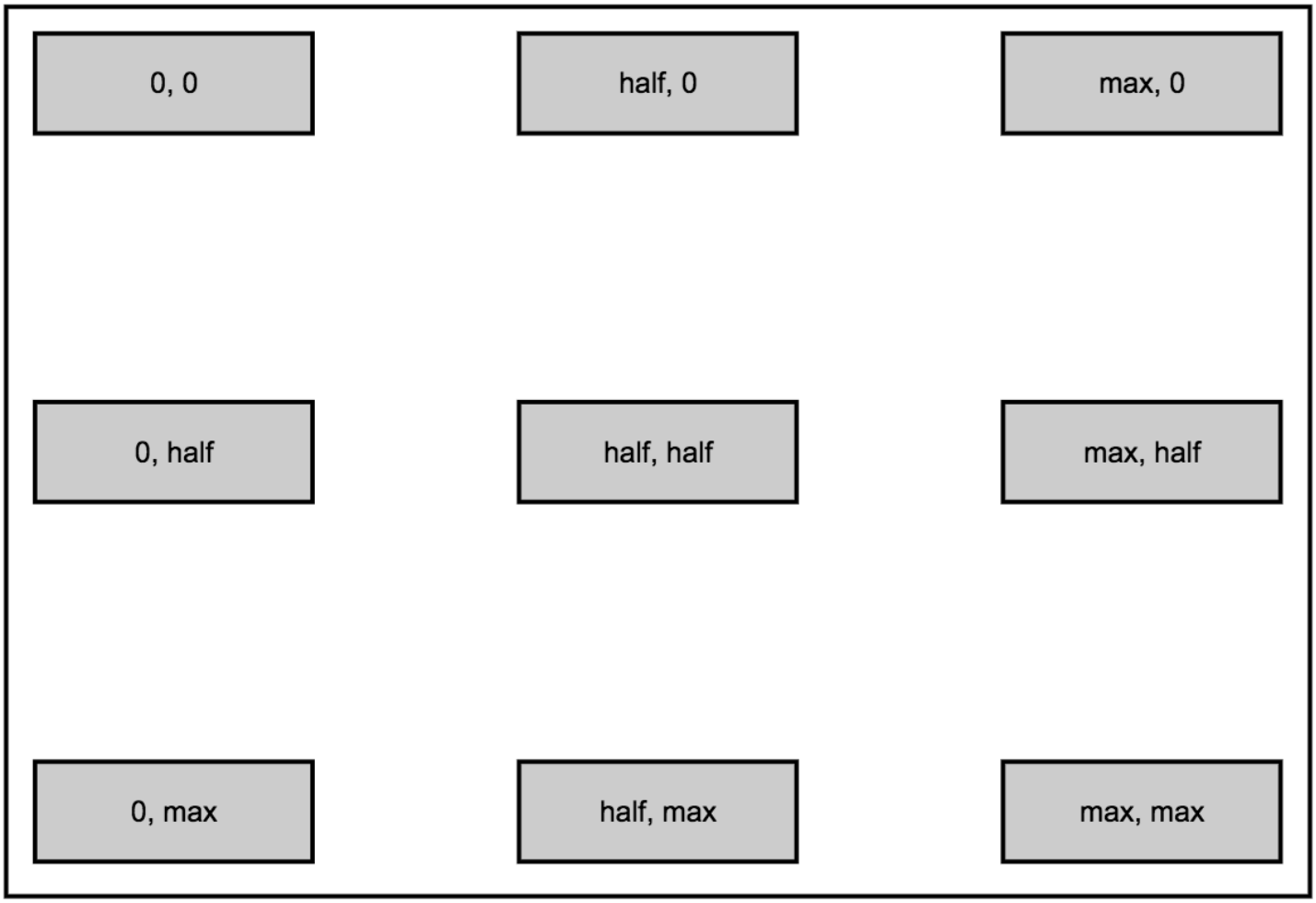

Figure 10: The 9 placement regions of the relative co-ordinate space.

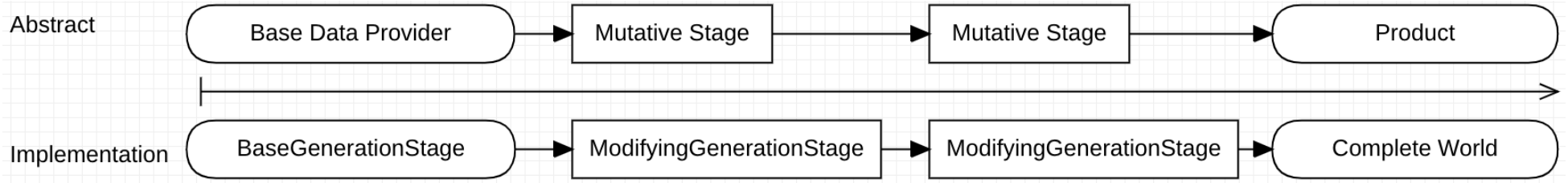

Figure 11: The design of the world generation pipeline.

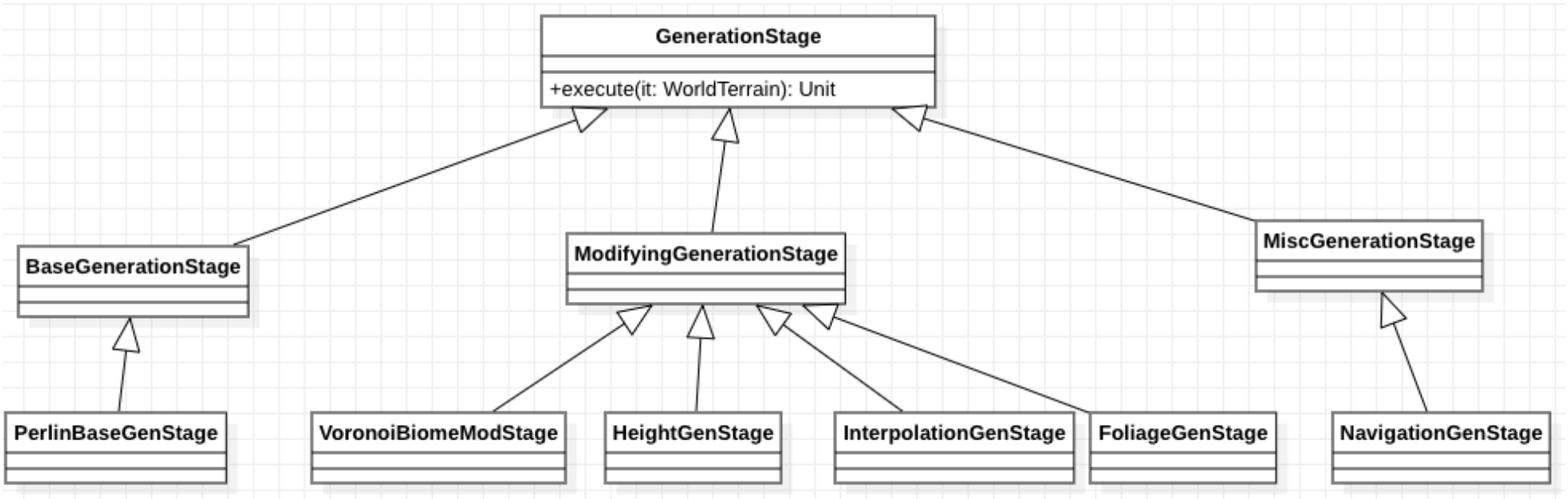

Figure 12: The design of the world generation hierarchy

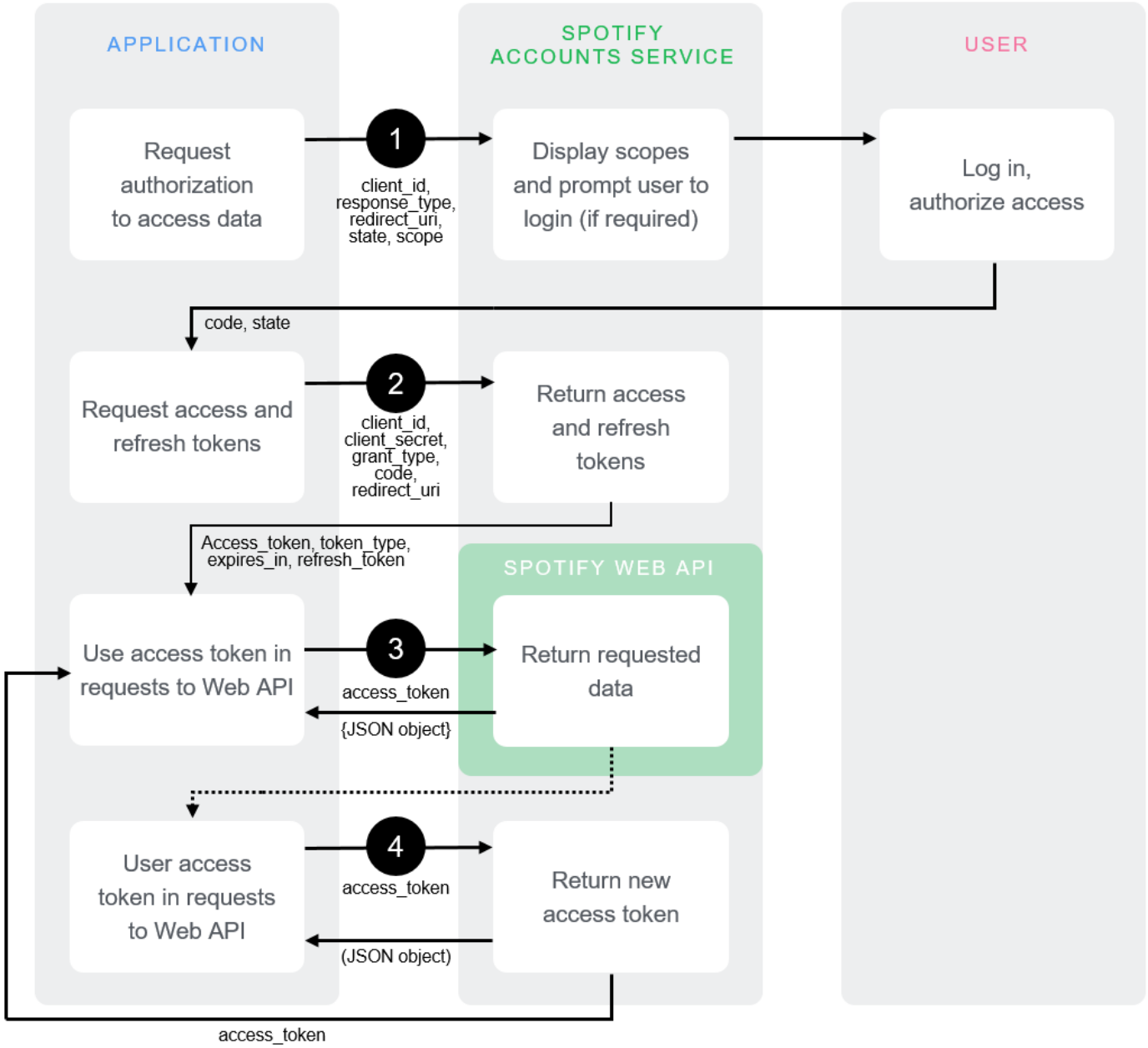

Figure 13: The Spotify authentication code flow.

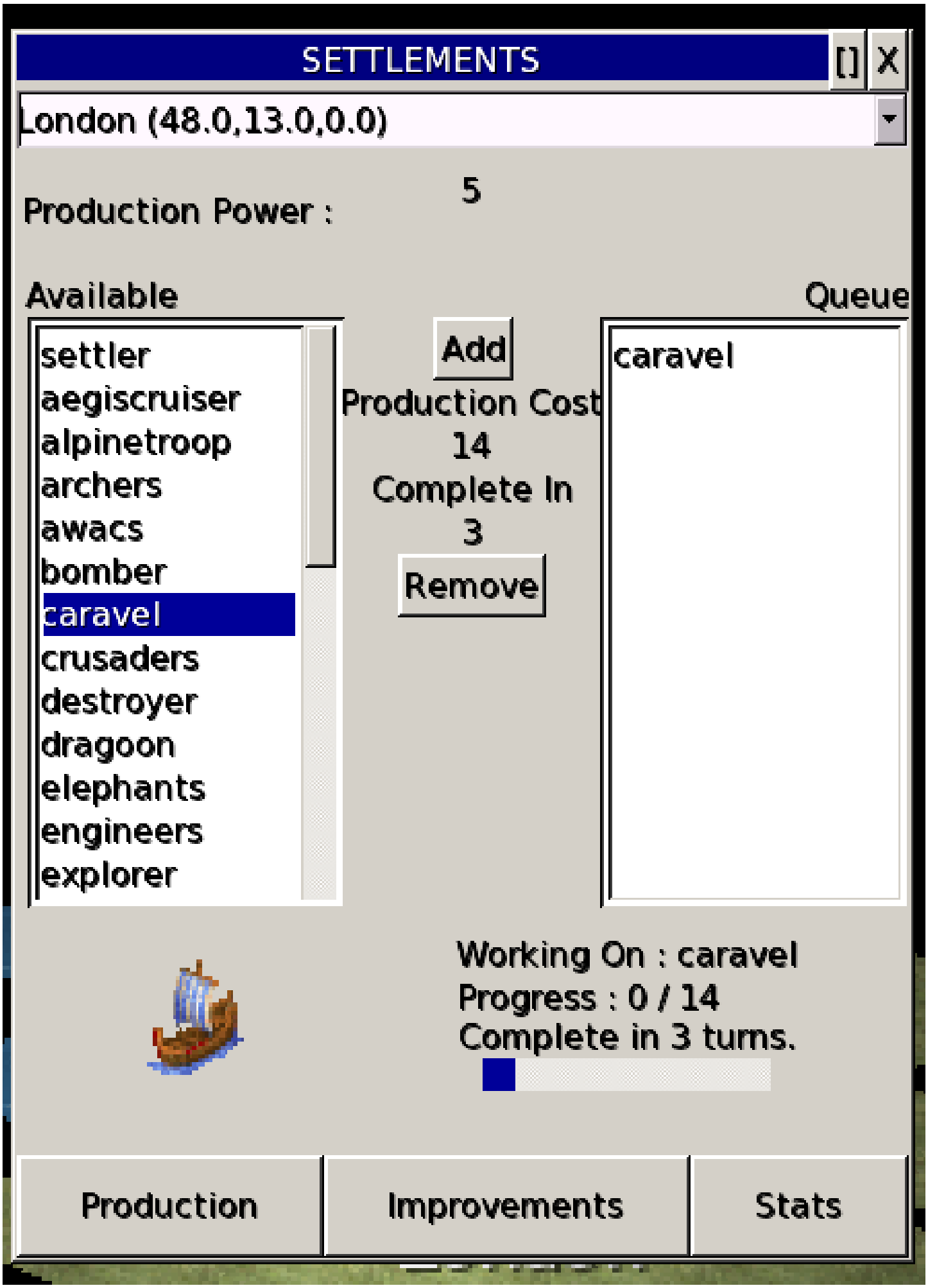

Figure 14: The UI pane used by the player to interact with a production manager. This one is from the Unit production manager.

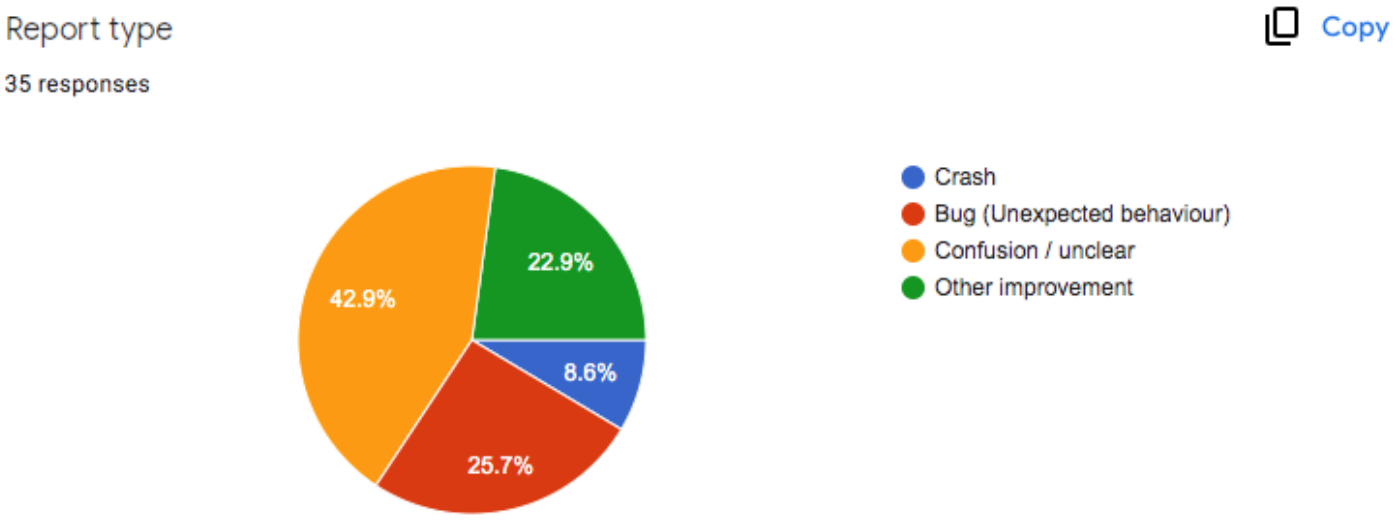

Figure 15: The percentage of feedback categories.

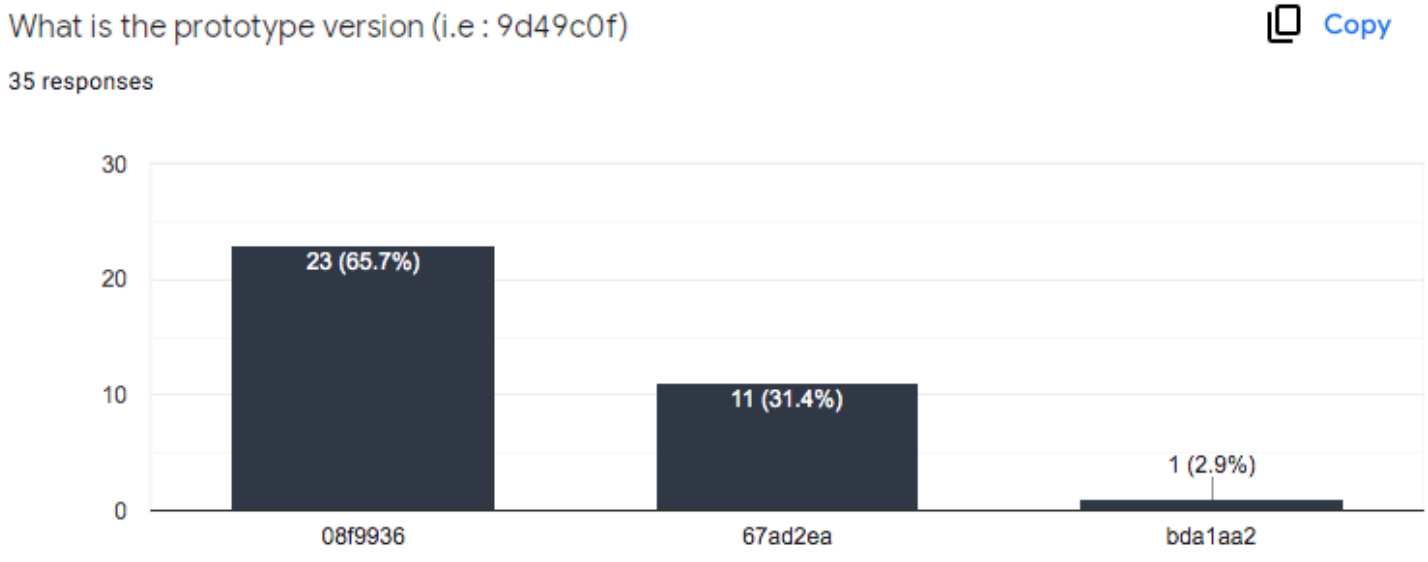

Figure 16: The distribution of commits tested.

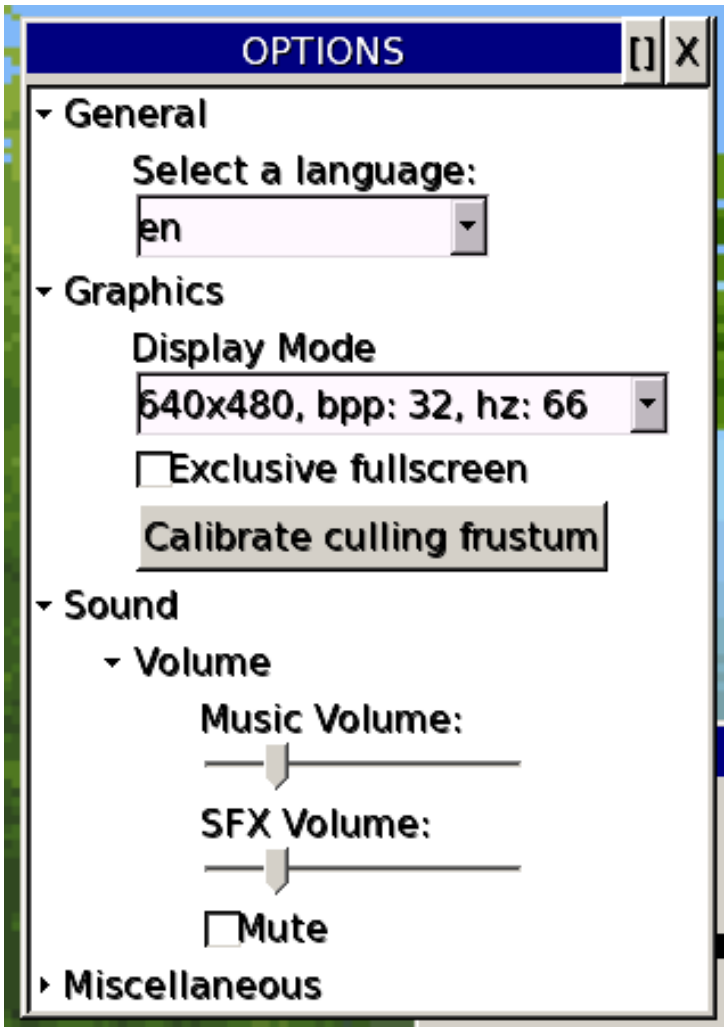

Figure 17: The new options window with a tree structure and reflective nodes.

Figure 18: The old options window with controls laid out in tabs.

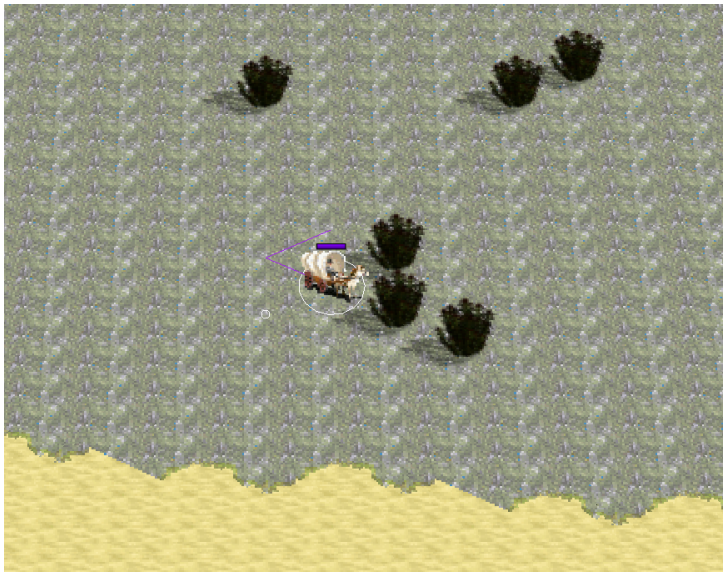

Figure 19: X6's terrain as rendered before fog-of war.

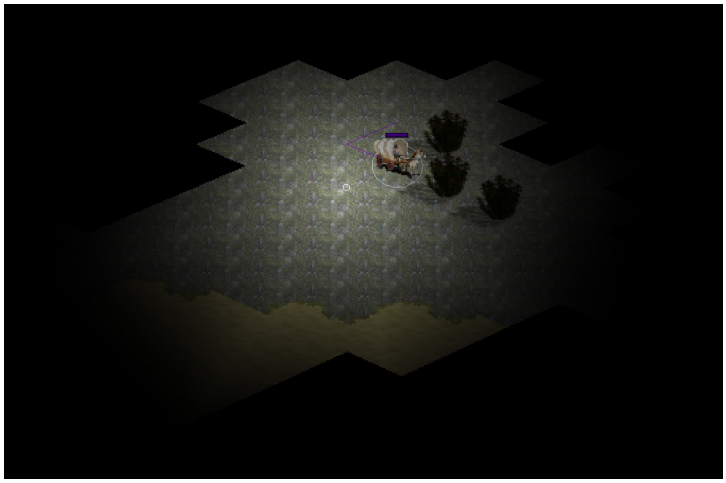

Figure 20: The same terrain as in the prior screenshot as rendered after fog-of war.

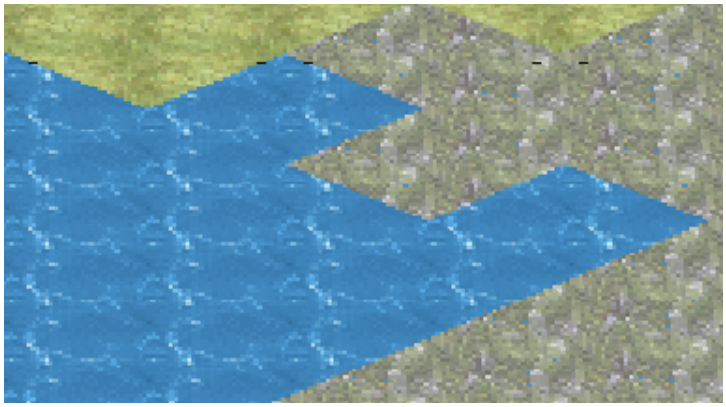

Figure 21: Some terrain tiles before being interpolated.

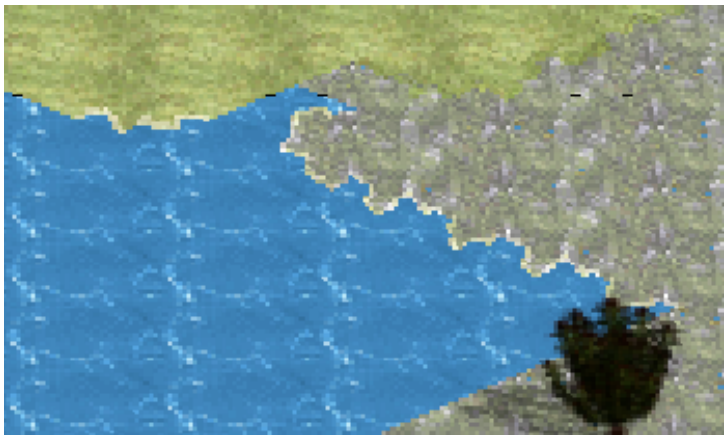

Figure 22: The same terrain tiles as prior screenshot after being interpolated.

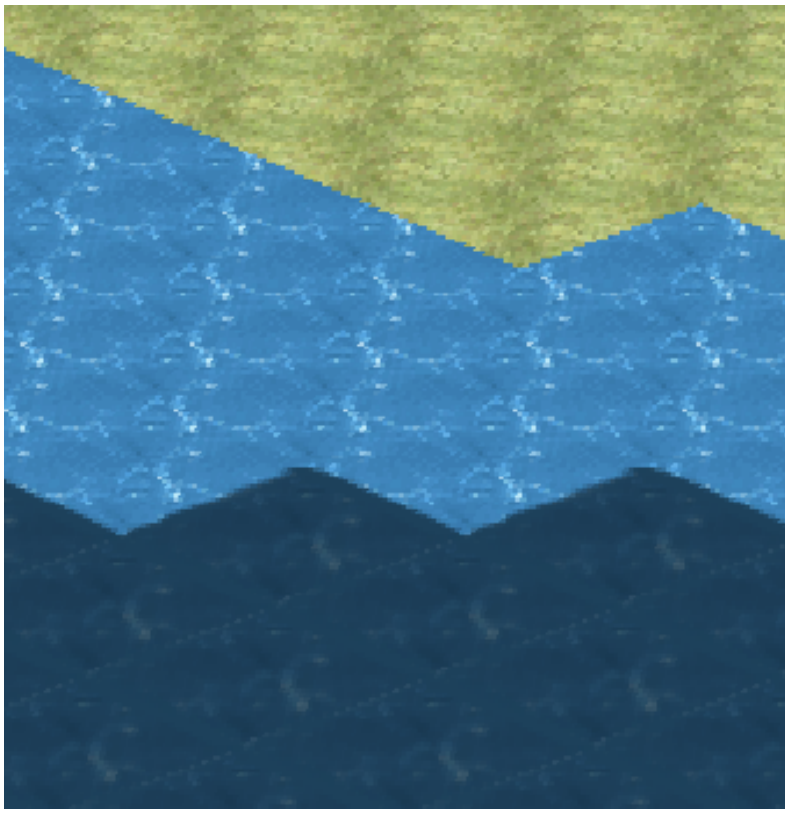

Figure 24: A deep beachline

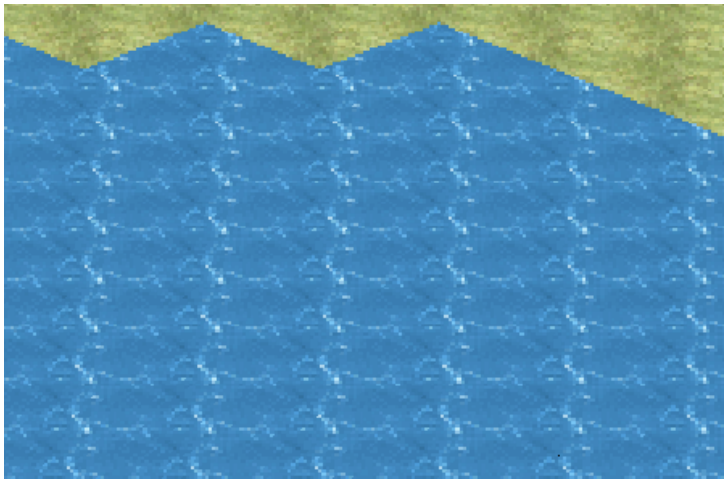

Figure 23: A shallow beachline

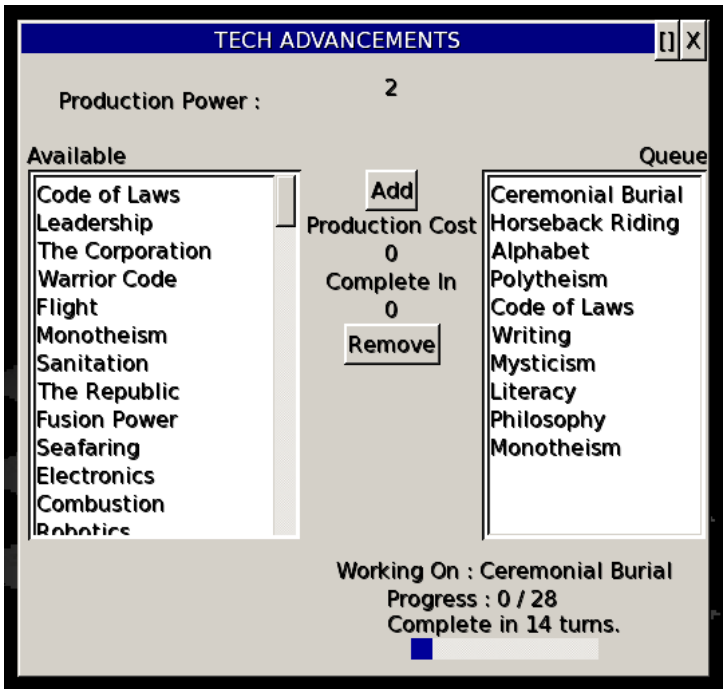

Figure 25: The production queue for the displayed advancement tree selection.

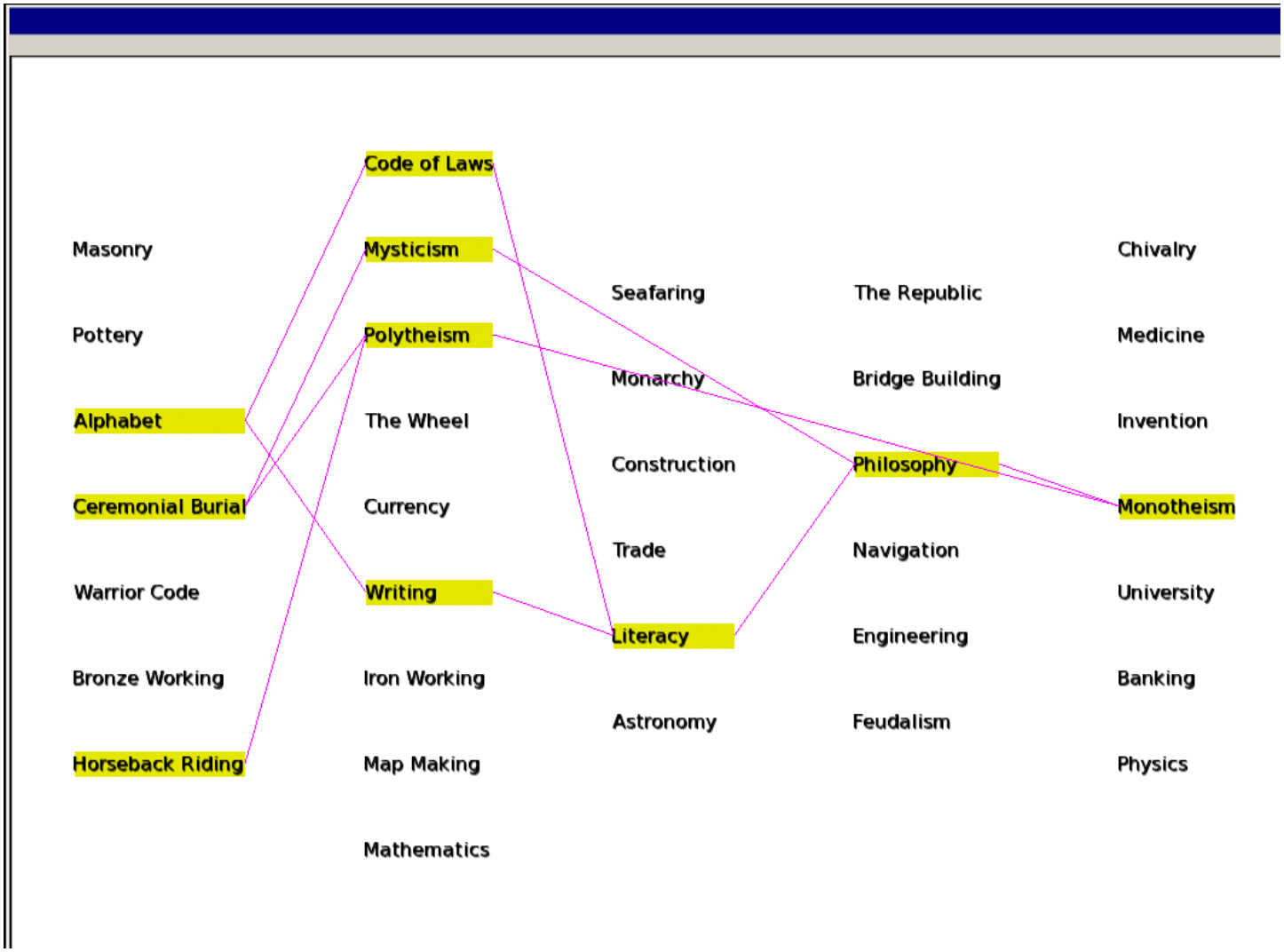

Figure 26: The advancement tree with 'monotheism' selected, and it's dependencies highlighted.

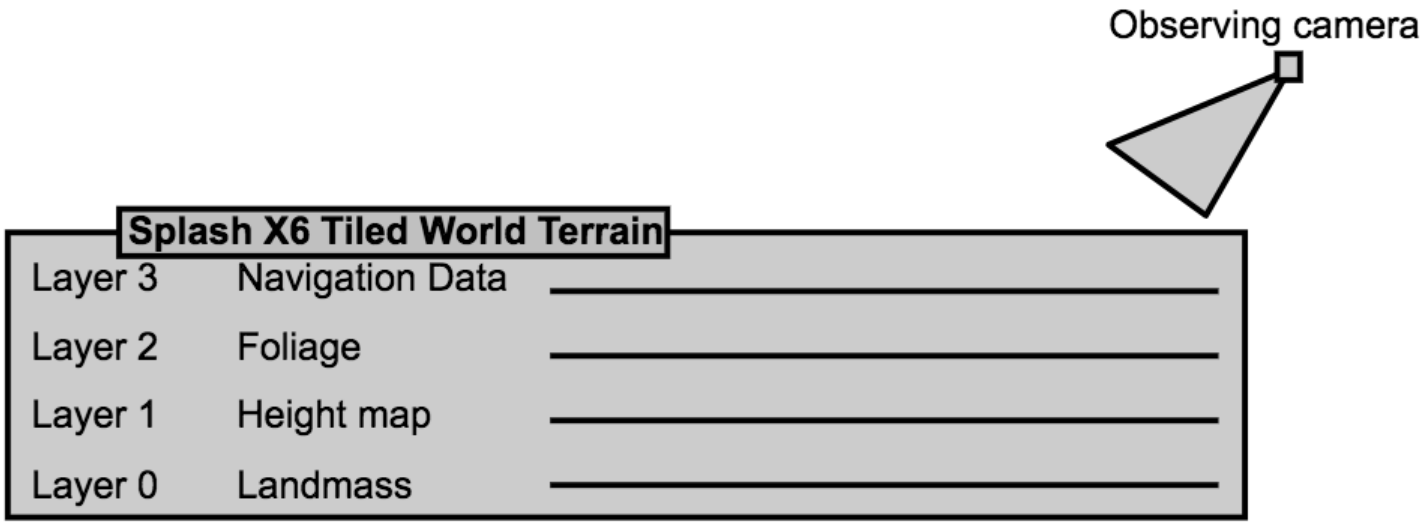

Figure 27: Layers of the X6's tiled world.

Glossary

.tmx Tiled Map. 5

10-Patch An extension of 9-Patch - An image with specific regions defines that may be stretched or repeated in order to resize an image whilst maintaining fidelity with borders and backgrounds in the texture.. 3

Adapter A blank implementation of an interface whose soul purpose is to reduce clutter in implementing classes, specifically where some of the interface methods are not required. The class extending the adapter may choose to override only the methods it requires, and is not forced to contain every method regardless of weather or not it will use it.. 4

ANTLR (Another Tool for Language Recognition) A language recognition system which uses a defined grammar to generate language tokenisers and traversers in java for text following that grammar.. 12, 19

CSS Cascading Style Sheet - Style sheet language used to describe how the contents of a markup document should be presented. 3

draw-able texture region atlas A single image containing multiple the visual texture resources, with an atlas defining the regions within the images at which each resource can be found. 3

drawables Any graphical image which may be drawn within the rendering environment, i.e a texture or font.. 3

Event Listener A script attached to an object which is ran when a specific event occurs involving that object. (i.e a click listener invoked when a button is clicked.). 4

FreeCiv An open-source re-creation of the original Civilisation games.. 8–10

FSM Fininte-State Machine. 11

https://itch.io An indie game and game asset repository / dispensary.. 3, 8

JSON JavaScript Object Notation - An open file format standardisation for storing and transmitting data in a key to value mapping format. 3

Letter Box Some rectangle placed inside another differently shaped rectangle such that the entirety of the inner rect is visible without deformation. This imposes void space on two sides, determined by the difference in aspect ratio between the two rectangles.. 4

LibGDX A Java game engine building upon industry standard wrapper bindings for low level API's, such as OpenGL, OpenAL, OpenCL & Box2D. 3–5, 9

LWJGL Light Weight Java Gaming Library - A java library with bindings for low level API's, such as OpenGL, OpenAL, OpenCL & Box2D. 3

OpenGL Open Graphics Library - A low level graphics API for hardware accellerated rendering.. 3

OpenTTD Open Transport Tycoon Deluxe - An open source remake inspired by the mid-nineties game 'Transport Tycoon'.. 3

OS Operating System - Acronym.. 3

reflection Meta invocation, observation and manipulation of code in the runtime. i.e code searching though a class for methods by name.. 4

RPG Role Playing Game. 3

Runnable A script which may be ran. Natively in java, the Runnable interface is intended for use with threading.. 12

Skin Composer A graphical utility for compiling visual assets with other visual data to form a LibGDX skin. 3

SMS State Machine Scripture - A custom scripting language for state machines.. 12, 19

Stage2D stage A 2D plane provided by LibGDX's Stage2D wrapper in which actors may be drawn. Actors may be related to 2D game (i.e sprites), or *stage2d.ui*'s GUI widgets.. 3, 4

Tiled A 2D level system supported by many game engines. Levels are pre-designed by hand in the Tiled editor.. 5, 11

Tree View A UI Widget containing nested *node* widgets used to represent a tree data structure of some kind. Also useful for grouping control widgets into branching categories.. 4

Unity A real-time 3D engine used for games, animation, and other 3D projects.. 11

web hook A local web API callback invoked by a third party to provide a notification of an event, or asynchronous response.. 8