Grid based java framework

Otto Zell & August Baaz

May 2021

Contents

1 Introduction **4**
- 1.1 Functionality requirements . . . 4
  - 1.1.1 Basic requirements . . . 4
  - 1.1.2 Additional features . . . 5
  - 1.1.3 Things that we did not have time for . . . 5
- 1.2 General requirements . . . 5
- 1.3 Similar frameworks . . . 6

2 Design **7**
- 2.1 Data structure . . . 7
  - 2.1.1 Tile . . . 7
  - 2.1.2 Board . . . 7
  - 2.1.3 Tabletop . . . 7
- 2.2 Interaction and dynamic behaviour . . . 8
  - 2.2.1 Entity . . . 8
  - 2.2.2 Observer pattern . . . 8
  - 2.2.3 Interaction . . . 8
- 2.3 Graphics . . . 9
  - 2.3.1 View . . . 9
  - 2.3.2 Window . . . 9
  - 2.3.3 CanvasComponent . . . 9
  - 2.3.4 Renderer . . . 9
  - 2.3.5 RealtimeRenderer . . . 10
  - 2.3.6 Tiles and boards . . . 10
  - 2.3.7 TextureHandler . . . 10

3 Testing **12**
- 3.1 Sokoban, and assertions . . . 12
- 3.2 Informal testing of graphics . . . 13
- 3.3 Automated testing . . . 14
- 3.4 Quick and dirty testing . . . 14

4 Important and interesting code **15**

5 Result **17**
5.1 Pictures 17
5.2 Did we manage to implement all functionality 18
5.3 Does it work as intended 18
5.4 Is it easy to extend functionality 19
5.5 What would we improve? 19

6 GIT **20**

References **22**

1 Introduction

As a final project in the course Advanced object-oriented programming (DT4014), we were tasked with creating a framework for the tile-based puzzle game [3]Sokoban. The framework should be as general and flexible as possible, and Sokoban should be considered a minimum requirement of the framework.

1.1 Functionality requirements

It's difficult to write code that you do not have a use for. To make a more general framework for a simple game, a good starting point is to instead make a framework to a more advanced game. We, therefore, chose to create a simple [1] Nethack (Dungeon crawler) clone

1.1.1 Basic requirements

The basic requirements of our framework needed to create a dungeon crawler game are the following.

1. Multiple things on a single coordinate. A creature should be able to stand on a pressure plate or item.
2. Large board with zoom and offset. The "camera" should be able to follow the player when it moves around in a board that is larger than the window it's displayed in.
3. Layered graphics. It should be easy to create graphics overlays that aren't bound to the grid.
4. Changing what is displayed. It should be easy to change what is displayed on the window, to a menu or an inventory.
5. Multiple ways to interact with the world. It should be easy to define multiple behaviours for interacting with the world. (walking, sneaking, pushing, kicking, observing and so on)
6. More than one moving entity. It should be easy to make the creatures of the world act on their own and move around when the player does.

1.1.2 Additional features

Things that are nice to have, which we implemented.

1. Realtime graphics. It should be easy to mix both real-time, and *on request* rendering.
2. Texture source tracker. It should be easy to import and use textures, without dealing with paths and file objects.
3. Dijkstra's pathfinding. Because that's awesome.
4. Procedural generation. Nethack is just not quite the same without it.

1.1.3 Things that we did not have time for

Things that are nice to have, which we would have added if we had more time.

1. Sound system. It should be easy to play up sounds without dealing with paths and file objects.
2. Serialization. Saving and loading objects or levels to file.

1.2 General requirements

1. The framework should be as general as possible, and be easy to extend further.
2. The framework should be written in java.
3. The framework should take advantage of object-oriented design.
4. The games we implement with the framework should showcase how flexible the framework is.
5. The entire project (framework, games, report) should be finished before the deadline of May 28th.
6. The project should not eat up too much of our time since we also need to study physics (but it did anyway whoops).

1.3 Similar frameworks

When it comes to similar frameworks we have to look way back in the history books to find anything that can even closely resemble our primitive framework. A framework we found that is also tile-based is called [4]Klik Play and was developed in 1994 by Clickteam, they seem to have similar functionality when it comes to being tile-based and the type of games it was designed for. One functionality that they have not incorporated in their framework is [5] Dijkstra's pathfinding.

It's difficult to find anything more modern since most modern frameworks are much more general 2D or 3D engines not constrained by a grid.

2 Design

The framework can roughly be split into 3 major parts. The data structure, the structure for dynamic behaviour, and the graphics. Holding these 3 parts together is a **PuppetMaster** object.

The **PuppetMaster** is the main game object of the framework. The is an abstract class that is extended to make a game. It does initialisation, contains the main-loop, and glues everything together.

2.1 Data structure

2.1.1 Tile

The smallest basic component of the framework is the **Tile**. A **Tile** object is nothing more than an object with a texture. Because the tile lacks coordinate it can be placed at multiple places at once.

2.1.2 Board

The tiles are placed on a grid in a **Board** object, which stores the tiles in a 2D array. The position of a tile on a grid is directly defined by its position in the 2D array.

2.1.3 Tabletop

With this basic structure, quite a few grid-based games can already be described, but a major limitation is that 2 tiles cannot share the same position. To solve this problem, boards are stacked in layers in a **TableTop** object. The tabletop stores one more boards in an array, and makes them easy to access.

2.2 Interaction and dynamic behaviour

2.2.1 Entity

An **Entity** object describes a unique **Tile**. **Entity** extends **Tile** with a coordinate. This is important for dynamic behaviour. It would be difficult to move a piece 1 step to the right if we do not know the current position. The coordinate is updated by the board to mirror its position on the grid whenever moved.

2.2.2 Observer pattern

The **Entity** class has an update() function, where time dependant behaviour can be added. This can be seen as a type of ***observer pattern***. update() called on a board, calls it on every entity on said board. The board keeps track of all entities in an ArrayList. This simplifies some bugs and improves performance when iterating.

2.2.3 Interaction

An **Interaction** object describes the interaction between a entity and a coordinate. Puppetmaster has a method called interact() which takes an **Interaction** object and a coordinate, and calls the action() method of the **Interaction** object.

This structure could be considered ***Strategy design pattern***, which describes an operation that an entity can make on the world.

It's done this way to make it as flexible as possible to define custom behaviour for Entities. Some sort of player or creature entity could have multiple behaviours when moved depending on what interaction object is used (walking, sneaking, pushing, kicking, observing and so on).

2.3 Graphics

2.3.1 View

A **View** object describes a graphics layout. It's an abstract class that can be extended to create any type of layout. A default button row where buttons can be added is part of the abstract **View** class, but doesn't need to be used.

2.3.2 Window

A **Window** object describes the visible window. A window contains one or more **View** objects, but only one view is visible at a time (can be changed by a method call). Making a menu that leads to a sub-menu is very simple.

Drawing is done on demand by calling the draw() function. The draw method will call draw() of the current view. the draw method can also be called directly on the view object.

The window will always keep the aspect ratio of the view constant, regardless of how the window is resized.

2.3.3 CanvasComponent

A **CanvasComponent** object, is a raster graphics area that can easily be added to a view. The **CanvasView** class is a default view that uses canvasComponent.

2.3.4 Renderer

a **Renderer** object describes what is drawn to a **CanvasComponent** object. Each renderer has a **TransformationContext** object, which takes care of scaling and offset automatically. Multiple renderers can share TranformationContext object, making it easy to draw overlay graphics regardless of zoom, scaling and offset.

Making a new renderer can be done by extending Renderer, and adding code to the abstract render() method. Adding multiple renderers to a single canvasComponent makes it easy to create layered graphics.

2.3.5 RealtimeRenderer

The RealtimeRenderer class extends the Renderer. The RealtimeRenderer has separate render() functions for real-time and non-real-time drawing. Mixing real-time and non-real-time rendering is no issue. The non-real-time renderers will still only render on demand.

The real-tile render loop is part of window. It can be started with a time interval of choice or stopped if needed.

2.3.6 Tiles and boards

The graphics structure of the framework is independent of what is drawn or what kind of game it is. But since this framework is designed for grid-based puzzle games, there should be some grid specific code.

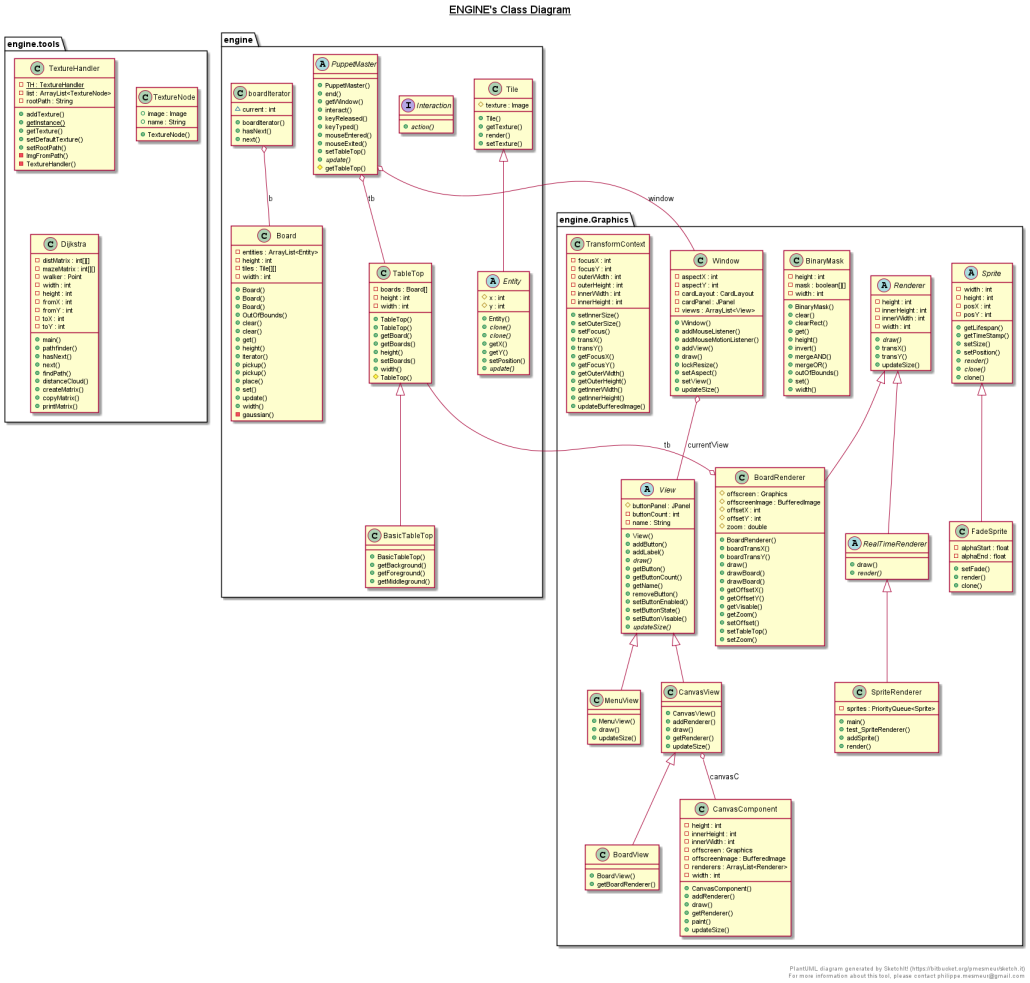
The BoardView class is a default view. It's in essence a CanvasView with a default renderer that draws a tabletop.

Each tile and entity has a render function. It will draw itself at a given coordinate, with a width and height of 100px. Overwriting this render method allows tiles and entities to have compound graphics. It can be a health bar, layered images, or a texture that changes based on a condition.

2.3.7 TextureHandler

The **TextureHandler** is a ***Singleton design pattern***. It keeps track of textures files. Reading a file from a path, and lading an image object each time you need a texture, is bothersome. This class makes it easy to load an image from a path and gives a nickname so it's easier to use in code.

The **TextureHandler** is used by the constructor of the tile class, to get **Image** object from a nickname.


ENGINE's Class Diagram
engine.tools
TextureHandler
TextureNode
Dijkstra
engine
boardIterator
PuppetMaster
Interaction
Tile
Board
TableTop
Entity
BasicTableTop
window
tb
engine.Graphics
TransformContext
Window
BinaryMask
Renderer
Sprite
View
BoardRenderer
currentView
RealTimeRenderer
FadeSprite
MenuView
CanvasView
SpriteRenderer
canvasC
CanvasComponent
BoardView

3 Testing

When coding a large project, it's important to test the code along the way to make sure what you write does what you intend it to. What kind of tests are appropriate depends heavily on what kind of code is being tested.

3.1 Sokoban, and assertions

The framework itself does not contain any behaviour code. Aside from graphics, which is tested informally. The rest of the framework is empty abstract functions or *setter and getter* code.

To test the behaviour code, we need something that actually has behaviour. [3]***Sokoban.java*** is very simple game built with the framework. To test that the game works properly beyond what can be observed from playing, assertions were added to almost every piece of code in the game.

This testing method could be expanded further, by writing code to automatically play the first level. But manually playing the first level yield the same result.

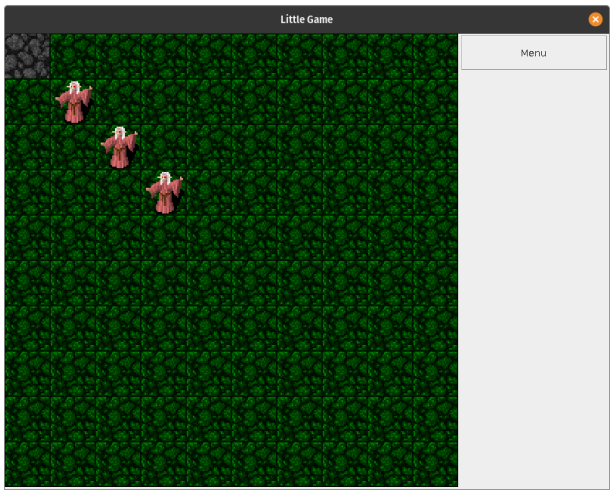
```java
public class MoveTo implements Interaction {
    @Override
    public boolean action(PuppetMaster p, TableTop tb, Entity e, int x, int y) {
        assert(p!=null);
        assert(tb!=null);
        assert(e!=null);
        assert(p instanceof Sokoban);
        assert(tb instanceof Level);

        boolean b;

        Sokoban game = ((Sokoban)p);
        Level world = ((Level)tb);
        Board fg = world.getForeground();
        assert(fg!=null);

        if(fg.OutOfBounds(x,y)) return false;

        assert((Math.abs(e.getX()-x)+Math.abs(e.getY()-y))==1); // only moves 1 step

        Tile t = fg.get(x,y);
        if(t==null){
            int testx=e.getX();
            int testy=e.getY();
            Tile e2 = fg.pickup(e);
            assert(fg.get(testx,testy)==null); // pickup successful
            assert(e2==e); // picked up correct object

            b=fg.place(e,x,y);
            assert(b);
            assert(fg.get(x,y)==e);
            assert(e.getX()==x && e.getY()==y);
            return true;
        }
```

3.2 Informal testing of graphics

The vast majority of code in this framework is for graphics, which is difficult to write automated tests for. A lot of our testing is therefore informal testing where we observe the behaviour of the code when it's running.

SimpleGame.java is a simple java program that uses the framework, to help test graphics. The tests it does are the following:

1. Button for changing view back and forth. To test if changing view work.
2. 9x9 sized board in a **BoardView**. To test if **BoardRenderer** work.
3. Tiles with transparent background. To test if stacking of boards work.
4. Scrolling to zoom in and out. To test if scaling and offset of **TransformContext** work.
5. Clicking to change a tile. To test if coordinate translation of **TransformContext** work.
6. Real-time animated sprite that that is created at mouse-Click. To test if real-time graphics work. And layered renderers work.



3.3 Automated testing

As part of Dungeon.java, we needed to implement Dijkstra's pathfinding algorithm to help creatures chase the player around. The code for this was an old piece of code we wrote for another course, which we adapted to fit our needs.

This is a perfect example where automated testing shines. It's a complex algorithm with a single input and an expected output.

The simple test involves 2 "mazes", and an expected path it should find in each. This test helped a lot when the original code was made and helped us again in this project when optimizing the old code without breaking it.

```
mazeMatrix
[4, -1, 8, 7, 2, 7]
[2, -1, 8, 9, 3, 5]
[1, -1, 1, 9, 4, 2]
[2, -1, 3, -1, 5, 9]
[9, 9, 8, -1, 1, 7]
[-1, -1, -1, -1, 7, 2]
[9, 2, 2, 5, 6, 3]
[1, 3, 3, -1, -1, -1]
[8, 4, 6, 8, 2, 5]

pathMatrix
[0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0]
[1, 0, 1, 1, 1, 0]
[1, 0, 1, 0, 1, 0]
[1, 1, 1, 0, 1, 0]
[0, 0, 0, 0, 1, 0]
[0, 0, 1, 1, 1, 0]
[0, 0, 1, 0, 0, 0]
[0, 0, 1, 1, 1, 1]

correct solution:true
```

3.4 Quick and dirty testing

Even without much test-specific code, testing the code you write is part of coding itself. Writing a piece of code without making sure it behaves correctly is an express ticket to "integration hell" later on. Much of this kind of testing is however quick and dirty prints or debug tools that are later removed in the final code.

The primary purpose of this kind of testing is to reduce time spent debugging later on.

4 Important and interesting code

there is a lot of code that has some quirk and one example of this in our code that is relevant to the course material is the Strategy pattern that we call **Interaction**. Interaction is an interface with only one abstract class called action. This interface is called by the **PuppetMaster** and used by the game by creating a new class that corresponds to a type of interaction. Examples of classes that implement this interface are the **Attack** class and **MoveTo** class. this implementation makes it very easy to create many different types of interactions between entities.

```
1 public interface Interaction {
2     public abstract boolean action(PuppetMaster p, TableTop
    tb, Entity obj1, int x, int y);
3 }
```

Interaction

```
1     public boolean interact(Interaction i, Entity e, int x,
    int y){
2         return i.action(this,this.tb,e,x,y);
3     }
```

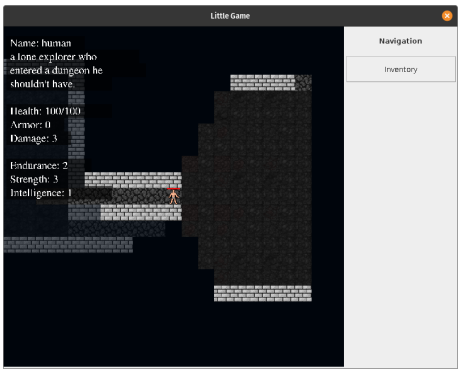
Implementation in PuppetMaster

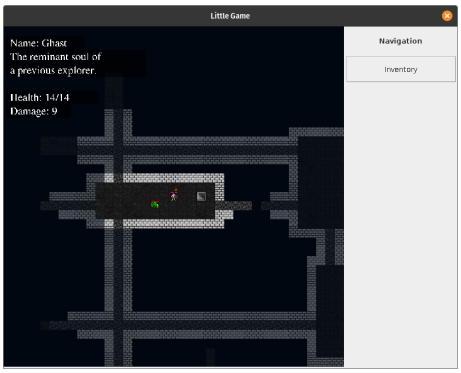
```java
public class MoveTo implements Interaction {
    @Override
    public boolean action(PuppetMaster p, TableTop tb, Entity
    e, int x, int y) {
        DungeonMaster game = ((DungeonMaster)p);
        Level world = ((Level)tb);
        Board back = world.getBackground();
        Board board = world.getForeground();
        Board floor = world.getFloor();

        if(back.OutOfBounds(x,y)) return false;
        Tile t = back.get(x,y);
        Tile c = board.get(x,y);
        if (e.getX()==x && e.getY()==y){
            return false;
        }
        // no walk more than 1 block
        if(Math.abs(e.getX()-x)>1 || Math.abs(e.getY()-y)>1){
            return false;
        }

        if(t instanceof MoveInto) {
            if (!((MoveInto)t).moveInto(game, e))
            return false;
        }

        if((c instanceof Creature)){
            Attack a = new Attack();
            a.action(p,tb,e,x,y);
            return false;
        }

        t = floor.get(x,y);
        if(t instanceof MoveInto) {
            if(!((MoveInto)t).moveInto(game, e))
            return false;
        }
        board.pickup(e);
        board.place(e,x,y);
        return true;
    }
}
```

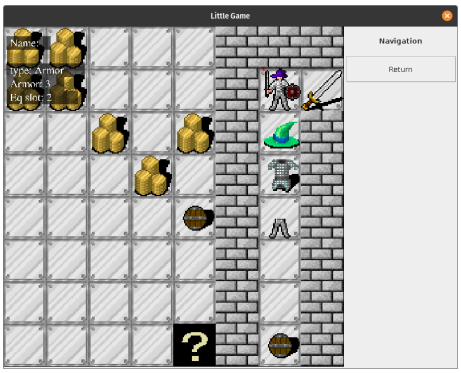
Actual implementation in a game

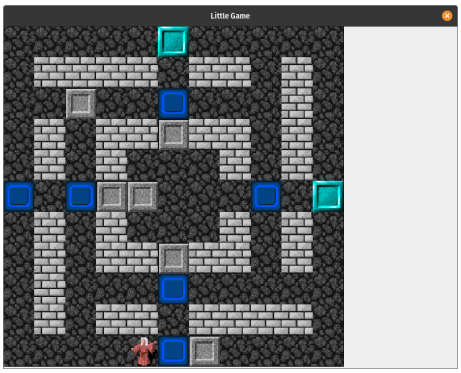
5 Result

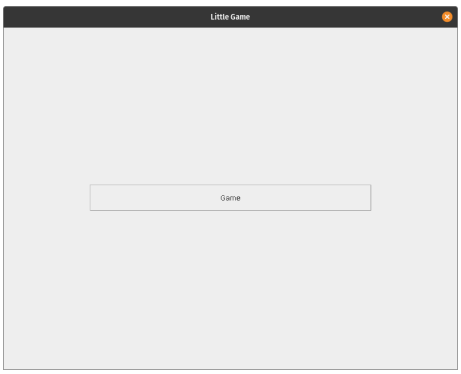
5.1 Pictures











5.2 Did we manage to implement all functionality

We managed to implement all the basic requirements specified in section 1.1. Of the additional features that we wanted to add, only sound system and serialisation are missing in the final framework.

The sound system would be somewhat simple to add, and we looked up the appropriate classes and libraries we need to get it working. However, we did not have much use for playing up sound, or any sound files to use. We, therefore, opted to skip adding it and focus on the report of what we already had instead.

Serialisation is a notoriously bothersome thing to code. It's not difficult to code, but it tends to be very sensitive to structural changes in the overall project. If we had time we would have added it, because it's very useful for this kind of framework. Writing it involves converting objects to and from JSON text, which java has a built-in library for.

We added real-time rendering at the very end of the project. We had planned to use it for spell effects and projectiles in the **Dungeon** game but did not end up using it in the game. We added it to **SimpleGame** as proof of concept instead.

5.3 Does it work as intended

Yes. It runs, it looks good, and it doesn't crash. The dungeon game is a tad unbalanced, but this course is not about game design.

5.4 Is it easy to extend functionality

Obviously, this depends on the type of game and what functionality you would want to implement. For most games that are in some way similar to dungeon crawlers or Sokoban adding functionalities or changing the GUI is no problem, this is shown further in the design part of this report. See section 2.3.4 for more information about extending rendering.

There are also things that our framework is inherently unsuited for such as making non-tile based games since that would require a whole different type of engine. The Graphics part of the framework would still be usable for other 2D games, but the rest of the framework would be quite useless.

A classic platform game such as super Mario would be feasible. To allow the player to be able to stand between tiles, one would require a [8] AABB collision detector, since the player would no longer strictly be locked to tiles.

Big maps are also a problem since the game will start to lag if we let it render too much. The board rendering code is not very efficient, and java AWT is difficult to hardware accelerate.

5.5 What would we improve?

The main big problem with our framework is performance, the games can't render more than 50x50 without lagging quite a bit, this could be optimized more by separating the game into chunks, a popular method particularly famous for its use in Minecraft, which coincidentally is also programmed in java, however this lag when big areas are rendered is an inherent problem with java graphics jfx and a big reason why all popular engines like [6] unity and [7] unreal use c++ or c#. Another improvement that would expand all graphical possibilities of our game would be to change the basic block from squares to triangles, this would make it easier to create less boxy 2d games. This however would require a rework of basically the whole engine.

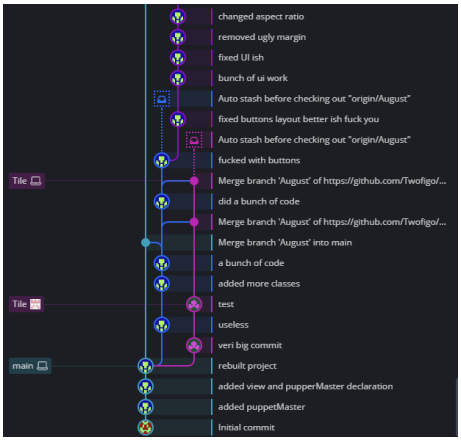
6 GIT

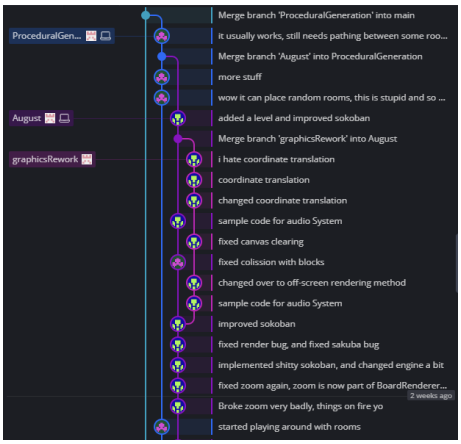
Git is a very popular type of version control that makes it easier to track changes in files. It helps us, the user, keep track of version and changes in their project. We used Gitkraken as our Git client, for its beginner-friendly UI that helps us keep track of how our repository is looking.

We had both used git before to track changes on personal projects, but we were both new to using git to collaborate on a project. We didn't have the best structure when we started coding with our git repository. In the beginning, our repository tree looked more like a spider web rather than an actual tree. However, after a week of working on it, we got more accustomed to using it together.

Our main use of git was to keep track of changes. It allowed us to make radical changes in code without risking the project. If we ever want to go back to an earlier branch we can simply do that.

We didn't use the collaborative parts of git much, we mostly used it to send our changes back and forth. In a way, we still treated the project as a personal project, where half the code wrote itself when you looked away. This method worked okay as long as we communicated well, but a more sophisticated approach may be needed when working with more than two people.





References

[1] NetHack is a open source single-player rogue inspired video game from 1987.

[2] A roguelike is characterized by 3 main things. 2D, Procedural generation with permanent death

[3] A puzzle game developed for computers in 1981

[4] Klick n Play is an engine with drag and drop functionality developed for 2d games with real time and turn based capabilities

[5] Djikstra's path finding algorithm

[6] Unity game engine

[7] Unreal game engine

[8] Axis-Aligned Bounding Box, a very simple collision detection algorithm