REDBEARD 2020 – OOP Project

Submitted by Akiva Stern

t.z. 320818008

**General Description:**

This project serves as a game “Redbeard”.

The game has multiple levels. In each level there is the main player – “**Redbeard**” , he is controlled by the left/right keys to run in that direction, and spacebar to jump.

The goal of the game is to find the **Gold** in each level.

In order to find the gold, the player must move around the virtual world seeking it, while he is able to walk/jump on walls, and colorful platforms (**Step**s). These colorful steps can move back and forth (right/left or up/down), or disappear leaving only the outline. In order to activate the qualities of these steps, the player must collect **Ball**s of the step’s color. These balls are located throughout the map. When the player spawns in each level, there is a **BulletinBoard**, stating the necessary amount of balls needed to collect in order to activate the steps of that color.

The player can only hold in his “**BallBag**” one color at a time, meaning if he has collected 2 red balls and then now collects a blue ball, the red balls will “fall” and return to where they were located originally in the map.

In some levels, there are also “**KeyTriangle**s” of a specific color. When collected, the steps of that specific color will remained unlocked regardless of the balls collected.

There are also obstacles that the player must avoid or it will cost him a life and he will respawn at the beginning of the level; moving **Monster**s, **Spike**s and rising **Water**. The water rises up (depending on the level) and can drown the player. If the player is submerged by the water, he has 5 seconds to “catch his breath” or he will drown.

In every level, there is a “Points Clock”. The faster the player finds the gold, the more points he can receive.

Good luck!

**Design:**

From the main, an object **GameController** is called to run().

The GameController runs the SFML window, and calls upon the Main Menu screen, from which the user can select to Play the game.

The GameController holds an object called **WorldOfObjects**. This will act as and manage the interactive objects in the virtual world. In a loop, the GameController takes commands from the user (movement of Redbeard), calls the WorldOfObjects to update() and calls it to draw().

At the start of each level, the WorldOfObjects is called to initialize (setup) the level. Files pertaining to each level (see below for more details about level data format) are read and using class **Factory**, we build each object in the file accordingly.

WorldOfObjects holds the game objects as follows:

A vector of pointers to the **StaticObject**s, these are the **GameObject**s that are static (non moving)– and do not need to be called upon to move() each iteration of the loop.

A vector of pointers to the **MovableObject**s is also held. These are the GameObjects that *are* called upon to move() in each iteration.

In addition WorldOfObjects holds a vector of std::weak\_ptr of **Step**s. Step is the object representing colorful platforms/steps, as mentioned above. From class Step, there is a derived class **DisappearingStep** – this posseses the quality to disappear too (thus not being able to be jumped on by the player). This vector of weakptrs is for the purpose activating/disactivating all the Steps according to the **Balls** being held by the player.

WorldOfObjects also holds the **Water** (a MovableObject Derived) and our main player, **Redbeard**.

In the update method of WorldOfObjects, we move the Redbeard (according to user keys), and iterate move() over the MovableObjects. We also are checking for collisions between all the StaticObjects and the MovableObjects, with Redbeard (we are using MultiMethods to handle collisions in **CollisionHandling**.h/cpp).

The Water does not need to check for a collision, rather it needs to check whether or not it *contains* the player, and therefore it is not included in the MovableObjects vector in WorldOfObjects, as it uses a unique method, and also must be drawn separately over all.

Each iteration, we will check if we won/lost etc. This info will be updated through the appropriate collision handling functions.

Each MovableObject contains a member **MoveHandling**. Using the Strategy design pattern, each MovableObject initializes his MoveHandling. Specifically, the **Monster** and the Step, share common moving functionality – they both move back and forth (or up and down) in a direction, at a given speed. That is why *their* MoveHandling will actually be derived from MoveHandling class - **BorderedMove**.

Redbeard’s MoveHandling is **KeyboardMove**, and should we wish to add more players that can be controlled by the keyboard into our game, they would also have KeyboardMove.

Water Object gets **WaterMove**, a unique movement (the water rises over time).

**GameObject**:

All the game objects essentially are built either by a Sprite or by sf::Shape (the Steps are rectangles and the Balls are CircleShapes).

Therefore, in order to divide this common functionality, again we use the Strategy design pattern;

Each GameObject holds a member **Drawable**. According to the actual type needed (sprite or shape) each class will actually initialize its drawable to be either a **SpriteShape**, a **RectShape**, or a **CircleShape**. The actual calling of the appropriate constructor from GameObject to initialize the Drawable is using a Template “middle” class **TemplateDrawable** – because even the Sprite and the Shapes have common functionality on “type T”.

**Files Created:**

* main.cpp – for calling GameController and then run().
* GameController.h/cpp – For Object GameController which holds all the game data and runs the window, deals with user input, etc.
* Animation.h/cpp – holds animation data for an object that uses sprite sheet.
* Ball.h/cpp – for object Ball.
* BallBag.h/cpp – for object BallBag, which the RedBeard holds, and deals with as he collects/needs Balls.
* BorderedMove.h/cpp – for class BorderedMove, which derives from MoveHandling and moves according to borders.
* Bulletin.h/cpp – for the BulletinBoard which displays the number of balls needed for each color.
* CircleObject.h/cpp – derives from TemplateDrawable which derives from Drawable, and can do functions on sf::CircleShape for the Ball (and the TriangleKey).
* CollisionHandling.h/cpp – for MultiMethods collision handling.
* DisappearingStep.h/cpp – for class DisappearingStep.
* DrawableObject.h/cpp – serves as base class for all drawables. Each GameObject holds this type of object.
* Factory.h/cpp – for design patter Factory, used to create game objects in nice clean generic way.
* GameObject.h/cpp – base class for all StaticObjects and MovableObjects as they have common functionality (and should we wish to hold them all in one data struct).
* Gold.h/cpp – for class Gold.
* Ground.h/cpp – for class Ground – (like a wall), in game.
* KeyboardMove.h/cpp – derives from MoveHandling, movement according to keyboard.
* KeyTriangle.h/cpp – for class KeyTriangle, which unlocked all Steps of that color.
* LevelDataDisplayer.h/cpp – for displayer level data such as lives, points, level number.
* MediaResources.h/cpp – Singleton class for all media – images and audio.
* Monster.h/cpp – for class Monster.
* MovableObject.h/cpp – derives from GameObject. Contains MoveHandling.
* MoveHandling.h/cpp – base class from different move types (explained above why base class used – for Strategy design pattern because Monster and Step use it, and possible adding more player).
* RectObject.h/cpp – does methods on sf::RectangleShape – used by Steps. Derives from TemplateDrawable.
* RedBeard.h/cpp – Our main Player 😊
* Spikes.h/cpp – for object Spikes
* SpriteObject.h/cpp – derives from TemplateDrawable, has functionality for sprites.
* StaticObject.h/cpp – base class for objects that do not need to be moved every iteration.
* Step.h/cpp - for class Step.
* TemplateDrawableObject .h/cpp – derives from DrawableObject. Holds type T, and holds common functionality for sf::sprite and sf::shape.
* Utilities.h – holds consts and enums.
* Water.h/cpp – for class Water.
* WaterMove.h/cpp – derives from MoveHandling. Intended for the movement of the water.
* WorldOfObjects.h/cpp – holds objects in the virtual world, and manages them when called upon by GameController.

**Main Data Structures Used:**

In WorldOfObjects, all the GameObjects are held.

This includes a RedBeard, Water and:

* **Vector** of MovableObjects. To be able to generically call them all to move each iteration.
* **Vector** of StaticObjects.
* (Additional **Vector** of weak\_ptrs to hold Steps).

Each iteration the Movables are called to move, and then ALL objects check collision (water checks containment) with Redbeard.

**Algorithms worth noting:**

Not specifically, however we are using the following design patterns:

-Factory

-Multi-Methods

-Strategy ( x2)

Also, the codes of each level (See below) are read into a map, and the maps value for each key will be sent to the “factory build” of each object.

**Bugs Known:**

None.

**Additional Comments:**

After my presentation with Michal and Amitai, the following changes were made.

- vector of pointers to Steps in WorldOfObjects was modified from shared\_ptr to weak\_ptr.

- in KeyboardMove, casting of Redbeard was modified from dynamic to static.

FILE FORMAT:

Each level consists of two file: “levelX.txt” and “levelXcodes.txt”.

In “levelX.txt”, the objects of the world are laid out according to the character symbol.

Some of the objects require more info than just their char symbols, so to their right, they will also contain a code. In “levelXcodes.txt”, next to each code will appear the additional info needed to build that object. \*\*

Additionally, the line “w X” must be added for X being the water speed in that level, and a character for each color in the game (‘b p r y’ for blue, purple, red, yellow).

And the line “o x y z w” for o being like a ball and x,y,z,w the color needed for each respective ball in that level.

\*\* For example in the levelX.txt file there may appear “+4”.

This means that we are building here a Step, and the Step details (speed, borders) can be found at line with key ‘4’ in levelXcodes.txt.

Note: Redbeard should be placed with a space between him and the ground for proper positioning.