The Game:
CASTLE SIEGE

1 Overview

The assignment asks to implement the game idea pitched in the previous practical. My game covers most of the main features described in the presentation, although some are not as polished as I first envisioned. The differences are mentioned at various points throughout the design section of this report.

The “Player’s Guide” provides a detailed description of how to play the game. It also explains how one could expand the game. A short version of the instructions can be found within the game itself. Simply start it and click “How to conquer?”.

In order to compile and run the program, simply execute ‘make’ from within the submission directory. The commands ‘make compile’ and ‘make run’ can be used separately in order to compile or run the code, respectively. The core.jar file is the Processing library used by my Java program.

2 Design

2.1 Title

I call my game “Castle Siege”, as it reflects the main component and featured gameplay elements.

2.2 Genre

2.2.1 Tower Defence – reversed

This game reverses the traditional “Tower Defence” genre by making the player – the aggressor. There is no strictly defined related genre, so I refer to it as “Tower Attack”. It is somewhat close to Real-Time Strategy (RTS), as the player needs to use a strategy to win and all actions happen in real-time.

2.2.2 Elements of other genres

The game also includes small elements of other genres. At the end of every level, the player is given a choice between “good” and “evil”. The information is presented in a textual form and the actions are defined with labelled buttons. The choices made influence the gameplay, all similar to how “Text Adventure” games work.

The movement and power of soldiers is similar to real-world “wargames” where the strength of a unit depends on the number of soldiers within it.

The implementation of a level is slightly reminiscent of “puzzle” games. Each level comes with a problem (a set of towers) and an allowance on resources (limited number of soldiers and spells). The player uses the resource to solve (destroy) the problem.

2.3 Player

The player is not visually represented on the screen. They are only perceived through their actions. In the implementation, the Player class defines the actions that can be taken. It is supported by the Play screen class. Interaction happens almost entirely with the left mouse button. Click to select a soldier type, then click again to spawn a unit of that type. Magical abilities are activated in a similar fashion. Choices of “good” and “evil” are done with a mouse click. The only recognized key press is “Backspace”, which is used to restart the current level.

2.4 Goal

The game features a sequence of pre-defined levels. The goal of each level is to lay siege on a castle and capture it (by reducing it to zero hit points).

2.5 Opponents

The opponents in this game are towers that defend the castle. Each level features a number of towers in pre-determined positions. Each tower targets a unit of soldiers on its own, independent of other towers. The first unit of soldiers that enters its range becomes the target. The tower makes attacks against it until it is dead, at which point, it selects a new target. If there is no enemy within range, it stays motionless.

2.6 Rules & Mechanics

2.6.1 Soldiers

2.6.1.1 Movement

The player “picks up” a type of unit from the lower right corner of the screen. Then “drops” it at an arbitrary location near the bottom end of the screen. These actions both happen with the left mouse button. A red line appears to indicate the range if the user hovers beyond the allowed space. These computations happen in the “Play” class (in the package “screens”).

Once a unit is “dropped”, the game spawns a unit of ten soldiers of the selected type. They walk forward (to the upper end of the screen) in formation (*figure 7*). They engage any towers along their way (*figure 1*). Combat between soldiers and defences takes place until one side is reduced to zero hit points. After that, soldiers continue marching forward. Soldiers have a field of vision, so they will move sideways and attack adjacent towers, if seen.

After a defending tower is destroyed, the soldiers go back in formation. This simulates regrouping after a fight. In an ideal implementation, units with less than half of their original amount of soldier would bond together to form one unit.

2.6.1.2 Types

There are two types of soldiers. They share a lot of attributes. The abstract class “Soldier” (in the package “units”) defines most of the functionality. Inheriting classes are required to implement a couple of one-line methods, provide sets of images, and set several constants. Both “Footman” and “Archer” classes are implemented in such a way.

Different types of soldiers have different attack speeds (implemented as milliseconds between two consecutive attacks), different damage for each attack, different attack ranges, and different movement speeds. Soldiers do not individually have health. A unit of soldiers has an amount of health dependant upon the type of soldiers in it. Footmen are considered heavily armoured and thus have more health than archers. Every time the health of a unit is reduced by a tenth of the original, one of the ten soldiers is removed. Thus, units with less health have less soldiers and deal less damage. This is similar to how some wargames work.

When a unit of soldiers engages in combat, the time between two attacks of a soldier is adjusted by a random value between -15% and +15% of the original time. This value is computed again after every attack. The purpose of this is to create a feeling of chaos while fighting. A group of soldier is never completely in synch when their lives are on the line.

2.6.1.3 Collision

Collision detection has been implemented but not widely used. If a soldier’s attack range is less than its height, then the soldier is allowed to attack only if it collides with its target. Ranged soldiers (such as archers) bypass this check. There is a method that detects collision between two soldiers but I could not find a way to resolve it in a reasonable amount of time, so that method is not being called.

Collision is detected on an image-level. Two images collide if they cover the same pixel. I have used PNG images extensively throughout my game, as they make better visual representation. I could not come up with an appropriate collision detection algorithm on my own. I found [1] online and used it as my basis. I changed some elements of its loop.

2.6.2 Defences

Defences are immobile and shoot at units of soldiers within range. There are two types of towers. Ballista towers deal small amounts of damage to a single unit. Catapult towers deal strong area of effect damage but are slow to reload.

Most of their implementation is shared in the abstract class “Defence” (in the package “levels.elements”). It makes addition of other types of towers easy to implement. An inheritor class needs to only define two one-line methods in it, provide a set of images, and set several constants. Case in point – the Ballista class.

The differences in two implemented towers are attack speed (implemented as time between two consecutive attacks), damage for attack, and health. Moreover, the “Catapult” class overwrites the default attack method, so as to implement Area of Effect damage. Implementing a new defence can use the Catapult as its base (instead of the “Defence” class) and adjust other parameters to create a new AoE.

2.6.3 Moral

My “Castle Siege” features a “moral” mechanic where a morality score is kept during game play. The actions the player takes influence the moral. The moral’s only purpose is to increase the chance that the player will meet a magical creature (see next section).

Destroying towers reduces moral, while sparing towers increases moral. Sparing the lives of soldiers and using troops sparingly, increases moral. Using spells pushes the moral in the direction of the spell. Denying help from a magical creature pushes the moral score in the direction opposite the creature.

Each level of the game features a unique moral question (*figure 2*). Its answer has a greater influence on the morality score. In an ideal implementation, that answer would also have a small immediate effect on the following level (such adding a special tower to attack, or slowing down the soldiers). Currently, picking a side and sticking with is awarded with new combat options. Staying neutral is not awarded in any way. The addition of small immediate effects would make neutrality a viable option, as it will reduce the chance of negative effects on levels.

2.6.4 Spells

2.6.4.1 Magical Creatures

My game includes Spells that can only be cast by magical Creatures. The player has a chance to meet such creatures at the end of every level, after answering the moral question (*figure 3*). Each magical creature can cast one spell. The creature describes its spell when the player meets it. If the player accepts the offer of the magical creature, they are given permanent access to that spell. If they reject the offer, their morality score is adjusted.

Magical Creatures can be good or bad. The chance to meet such a creature depends on moral. High moral means a better chance of meeting good creatures. Low moral means a high chance to meet evil creatures. In an ideal implementation, creatures would also give access to new types of soldiers called “champions”. A champion would essentially be a unit of one soldier with very high attribute values. It could also overwrite some methods to implement multiple attacks over the course of an animation cycle.

2.6.4.2 Casting

To cast a spell, the player has to select the spell from the bottom left corner of the screen, and then select the area to be affected. This is similar to how soldiers are placed on the screen. There is, however, no restriction on where the spell is cast.

When hoovering, the range of the spell can be seen (*figure 6*). Different spells affect different parts of the battle. After being “drop” with the second mouse click, the creature casting the spell moves on to the screen, takes a couple of seconds to cast the spell, and moves out of the screen. Currently, this happens with an enlarge image and across the whole screen, with a freeze in the middle. In an ideal implementation, there would be animation taking place instead.

2.6.4.3 Effects

There are four implemented spells, each associated with a creature. The angel can cast “Heal”, which restores a lot of health to all units of soldiers within a wide range. The priest can cast “Haste”, which temporarily increases the movement and attack speeds of all units within range. The warlock can cast “Slow”, which temporarily decreases the attack speed and damage of all towers within range. Finally, the “Reanimate” spell, cast by the necromancer, creates zombies. A zombie is a type of soldier that can not be accessed otherwise. They are though and powerful but very slow. A separate unit of zombies is created for every unit with dead soldiers.

It is theoretically impossible for a player to have access to both Reanimate and Heal. However, as a precaution, I have implemented a method which prevents units from being healed after that method is called. If dead soldiers of a unit have been reanimated, then that unit cannot recover those soldiers any more. Also, zombies, themselves, cannot be healed. They also do no leave any dead for a follow up Reanimate.

In order to allow for temporary status changes, I created an Attribute class which contains a base value and a linked set of modifiers. A Modifier is, in turn, a description of how a value is affect and for how long. Whenever the value of an attribute is retrieved, all expired modifiers are removed from the set, then the remaining ones are applied. Modifiers are applied in the order they were added to the linked set.

Note: if there is trouble seeing the powerful spells in action (Heal, Reanimate), it is probably because of the random values. To increase the chance: open “assets/config/game.props”, find the line “footman spared moral change=15”, replace 15 with 1000, restart the game (no recompile needed), play through level one and spare one footman. This will set the moral score to 1000 for sparing one footman and thus, there will be a high probability of a meeting a good creature. Replace 15 with -1000 for a high probability of an evil creature.

2.6.5 Levels

The game has a set of predefined levels. A level is simply an arrangement of towers in front of the castle. It also specifies a limit on how many soldiers the player can use during the level. This gives it a puzzle-ly feel, as complex levels require more thinking.

The implementation of my “Castle Siege” makes adding new levels very easy. The core functionality is shared in the abstract class “ALevel”. Inheriting classes need only define the question at the end of the level, fill the collection of towers that protect the castle, and set some limits as integers in two maps (from soldier type to amount remaining, and from creature name to amount remaining). Upon creation, each level has two defence towers at start. These can be removed if needed.

In an ideal implementation, levels would be defined in text files and each level would have its own image for castle and background. The Java code would then parse those and create levels dynamically. Currently, levels store a reference to the class of the next level (via reflection). This avoids object creation of all levels early in the game.

2.6.6 Animation

Animation is strongly present in my game. The movement of soldiers, their attacks, and the attacks of the towers are all drawn with images. The images change over the course of a move/attack period. Images for an animation are stored in a folder and named “0.png” to “*N*.png” where *N* is the index of the last image.

Implementing the animation loops was not “technically challenging”. However, finding images, cutting them out, and putting them in order took time. My peers have told me that my game looks good and that it is satisfying to look at the soldiers’ movement.

Most images were taken from The Spriter’s Resource[9]. The entry in the bibliography gives all links to images used.

3 Context

I have not been been able to find many similar games. My default comparison is with “Kingdom Rush”[2], which is a popular Tower Defence game. The player uses towers to defend against invading soldiers. My game does the opposite, where the player uses soldiers to attack defending towers. “Kingdom Rush” and my “Castle Siege” are similar in that they feature different types of soldiers and towers. My game differs from “Kingdom Rush” thanks to its “moral” mechanic and the player’s choices influencing the options available in during siege.

Few games have the same concept. [3] has an “Angry Birds” style of play where the player destroys a castle. It is more of a puzzle than anything else. [4] implements the castle siege concept in the form of a card game. The game is played against an AI opponent. The cards can be used to spawn minions and cast spells similar to my design. However, the player also has to defend their own castle from the opponents attacks, whereas complexity in my game is added through a limited amount of resources (soldiers and spells). [5] focuses on wielding siege machines and the history of sieges in Scotland. My game lacks the historical aspect, but siege machines were on my list of desired features. Given more time, I would have implemented them as selectable unit types.

“Villainous”[6] is close to what I imagined, but there is no castle (at least at first) and it lacks moral choices. That game also starts off by intentionally loosing, which to me is both a disappointment and a drawback. The player can cast spells and send a limited amount of minions to battle, similar to my design. My game differs in that the soldiers can destroy the towers on their way. Furthermore, in my game, the player is not necessarily a “villain”.

Age of Empires: Castle Siege[8] features big castles, multiple soldiers, resources, and building construction. Similar to my design, there is no strict path that the soldiers follow. This is also a notable difference from games such as “Kingdom Rush” and “Villainous”. Age of Empires: Castle Siege lacks the moral choices after battles, and does not feature magic. Additionally, it also does not support collision handling between soldiers, which I assume would require great computational resource, given the capacity of the game.

Overall, few games implement the game idea where the player conquers castles and makes choices that result in different gameplay options. Similar games expand the combat while the distantly related RTS games add a second castle for the player to defend. The addition of moral choices in a

“Text Adventure” manner is what sets my game apart from others in this small genre. Spells are not unique to my game but are rarely found.

4 Evaluation

I tested my game by running it multiple times, placing my soldiers differently and selecting different options in the “Text Adventure” interactions. I tested putting soldiers near the edges of the screen and that helped my find bugs in my “formation” algorithm that recalculates the positions.

Multiple people played my game and the general feedback was that it looks good, and is fun to play, although it is easy to win. This relates to the feedback on my presentation that said that my game should not have a “one winning strategy” where placing all soldiers and rushing in one line wins the game.

My submission includes small written feedback from several fourth year students. They are in “feedback/”. Two were received before implementing moral scoring via sparing soldiers and destroying towers. One was received after doing so but before a Player’s Guide was written.

The last person to test ended up with a fairly neutral morality at the end. They selected the good options in the text but used all of their soldiers to attack (and so didn’t get moral points for spearing them). This highlights the problem of neutrality being a weak option. After that, I updated the moral system, so it is easier to swing to good. I increased the moral values for some soldiers and added values for the magical creatures.

Based on all evaluations, I added spell descriptions to what creatures say when a player meets them, I described the controls more, and I increased the difficulty.

5 References

[1] Pixel Level Collision Detection
https://www.openprocessing.org/sketch/149174

[2] Kingdom Rush
http://www.kongregate.com/games/Ironhidegames/kingdom-rush

[3] http://www.arcadespot.com/game/castle-siege/

[4] http://www.kanogames.com/play/game/castle-siege

[5] https://gridclub.com/activities/castle-siege

[6] Villainous
https://www.newgrounds.com/portal/view/576124

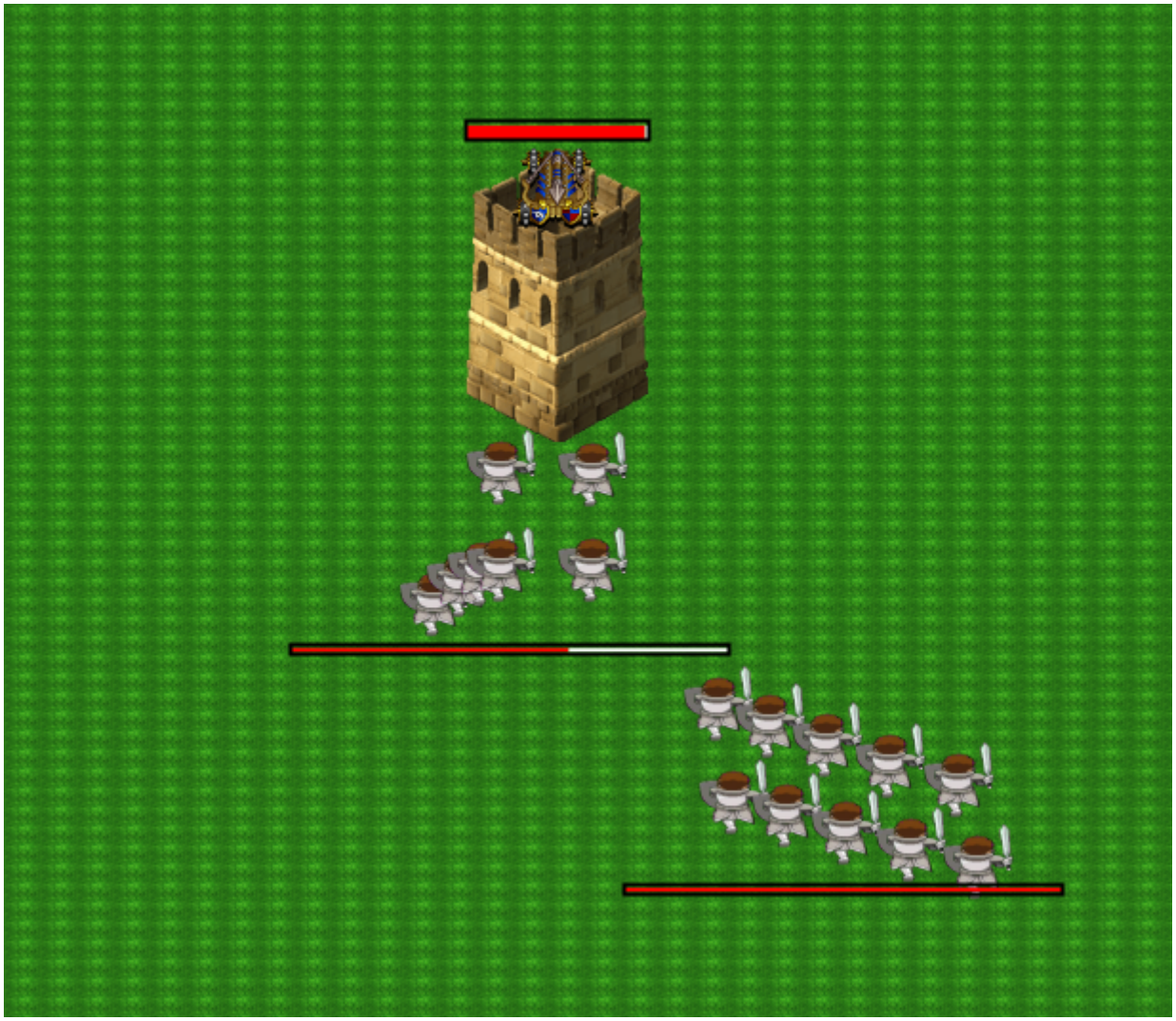
[7] Turret Tower Attack
This is actually a Tower Defence game but it came up in my search
http://www.dedegames.com/turret-tower-attack-2.html

[8] Age of Empires: Castle Siege
https://en.wikipedia.org/wiki/Age_of_Empires:_Castle_Siege
https://www.youtube.com/watch?v=WPXJIZzjRfs

[9] The Spriters Resource
https://www.spriters-resource.com/
https://www.spriters-resource.com/pc_computer/mightmagicclash/ [Soldiers and Spells]
https://www.spriters-resource.com/pc_computer/warcraft2/ [Ballistas and Catapults]
https://www.spriters-resource.com/pc_computer/zombieisland/sheet/59723/ [Tower]
https://www.spriters-resource.com/snes/dunquest/sheet/28538/ [Castle (unedited)]
https://www.spriters-resource.com/playstation/talesofphan/sheet/32763/ [Wooden castle gate]
https://www.pinterest.co.uk/pin/482025966348716562/ [green grass (background)]

6 Appendix

Figure 1. Shows a “damaged” unit of soldiers, which has less health than a “healthy” unit. Also shows that soldiers have seen the tower and move to attack it.



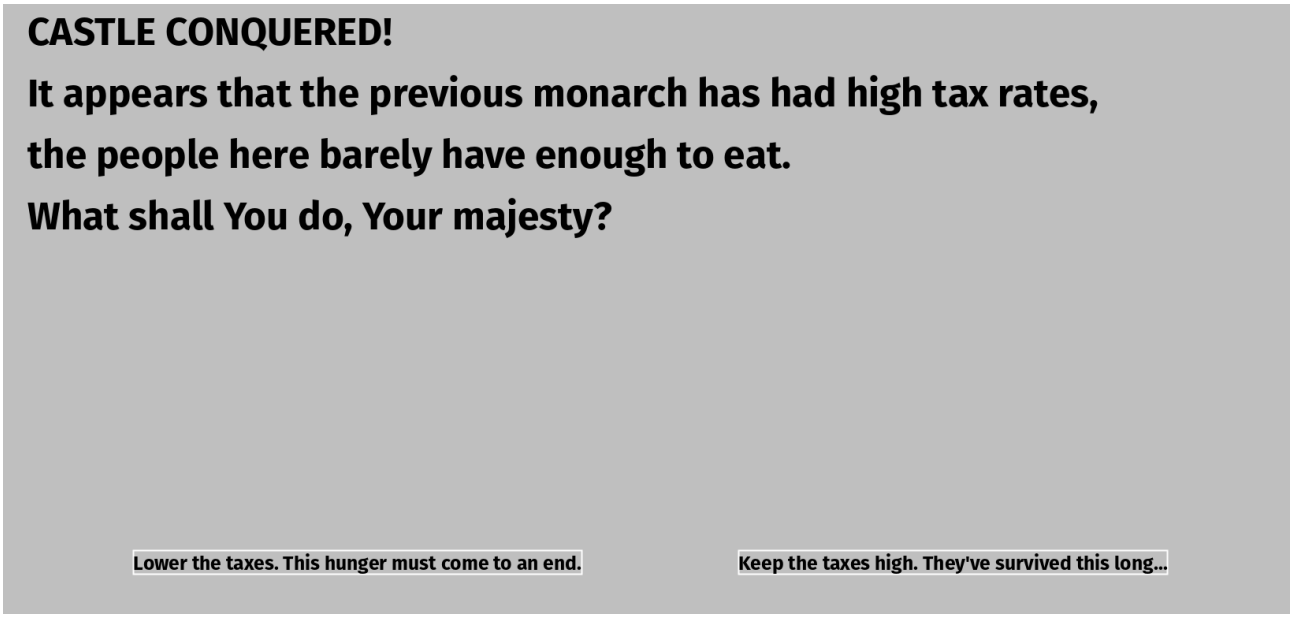


Figure 2. Shows a sample question after the end of the level.

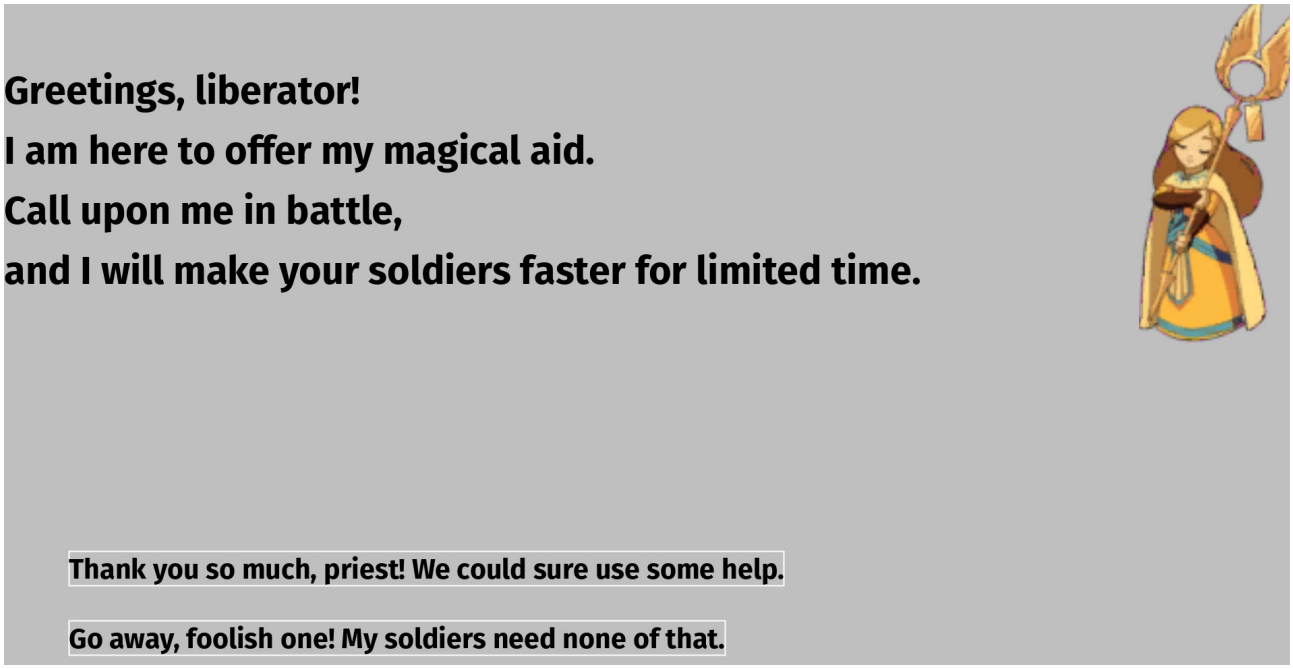


Figure 3. Shows a magical creature (“priest”), offer help.

Figures 4 and 5. Show the limitation both soldiers and magical creatures.

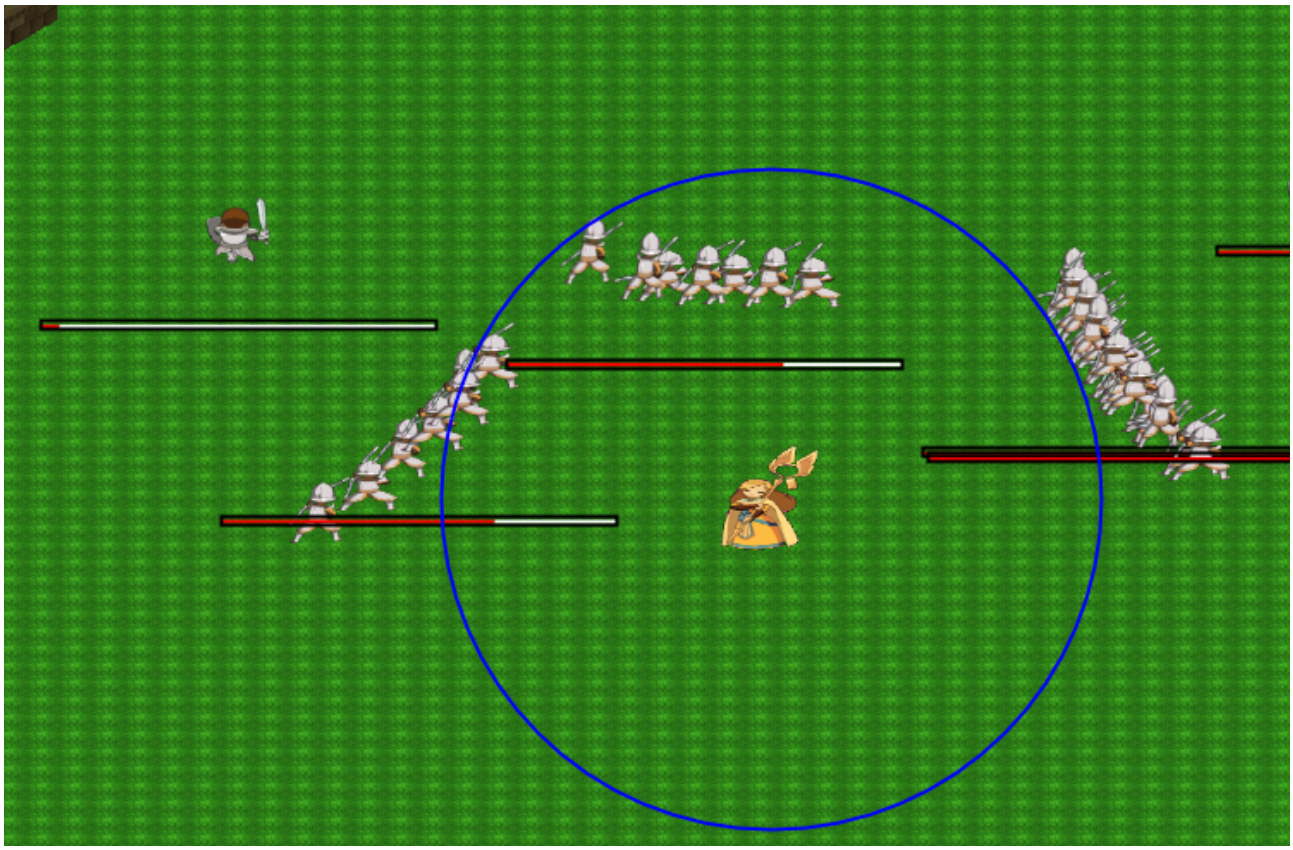


Figure 6. Shows the range on the priest’s Haste spell.

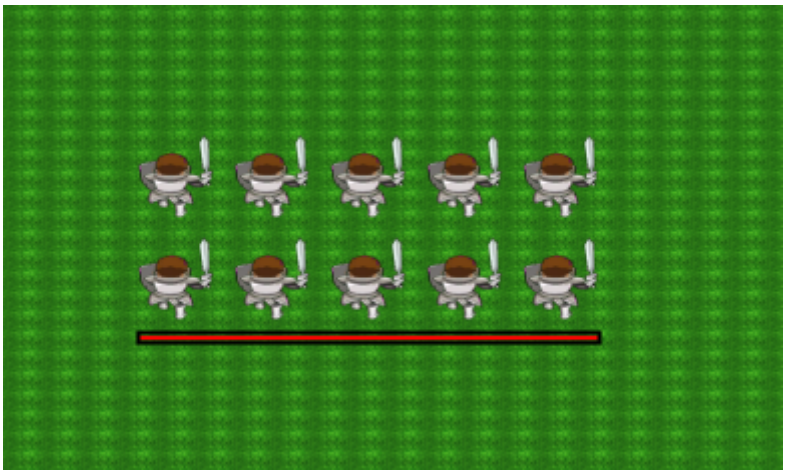


Figure 7. Shows soldiers walking in formation.