Revenge of the Moon - The Gathering Storm

Subject 2 - Minebombers clone

Authors:

- 84745F Miro Nurmela miro.nurmela@aalto.fi
- 290658 Henri Niva henri.niva@aalto.fi
- 84312L Joonas Lipping joonas.lipping@aalto.fi
- 78740E Roope Savolainen roope.savolainen@aalto.fi

Last update:

2013-01-11 21:22

Specification of requirements

All minimum requirements will be implemented. The game will feature single player and hotseat multiplayer, fog of war on the map, a store, collectible treasures, several game rounds, a few different terrain types (such as tar, ice, and walls of varying strength), a random map generator, and possibly some kind of procedural map generation and a continuous "survival" mode. The single player mode naturally comes with some simple AI enemies. Nothing special here — the game will not amount to much without these basic features. The most important goal is for the game to be fun to play, which can hopefully be achieved by enough testing and polish.

As for the optional requirements we were planning the following. Configurable keys and audio should be easy to implement - audio can be supplied through SFML (more on external libraries in the end of the documentation), and configurable keys can be implemented with simple menu selections and the like. "More intelligent AI" is somewhat vague as a requirement, but we do plan to implement more refined AI than just homing towards the player, for example. Sophisticated AI also allows us to provide AI players in multiplayer mode. Basic implementations of weapons and terrain are designed to be flexible, making it easy to add new ones. Campaign and map editors are in the planning and should not be hard to implement (not to mention that making maps by say, writing text files sounds like a terrible idea). We hope that our final map generator fulfills the criterion of being "super cool".

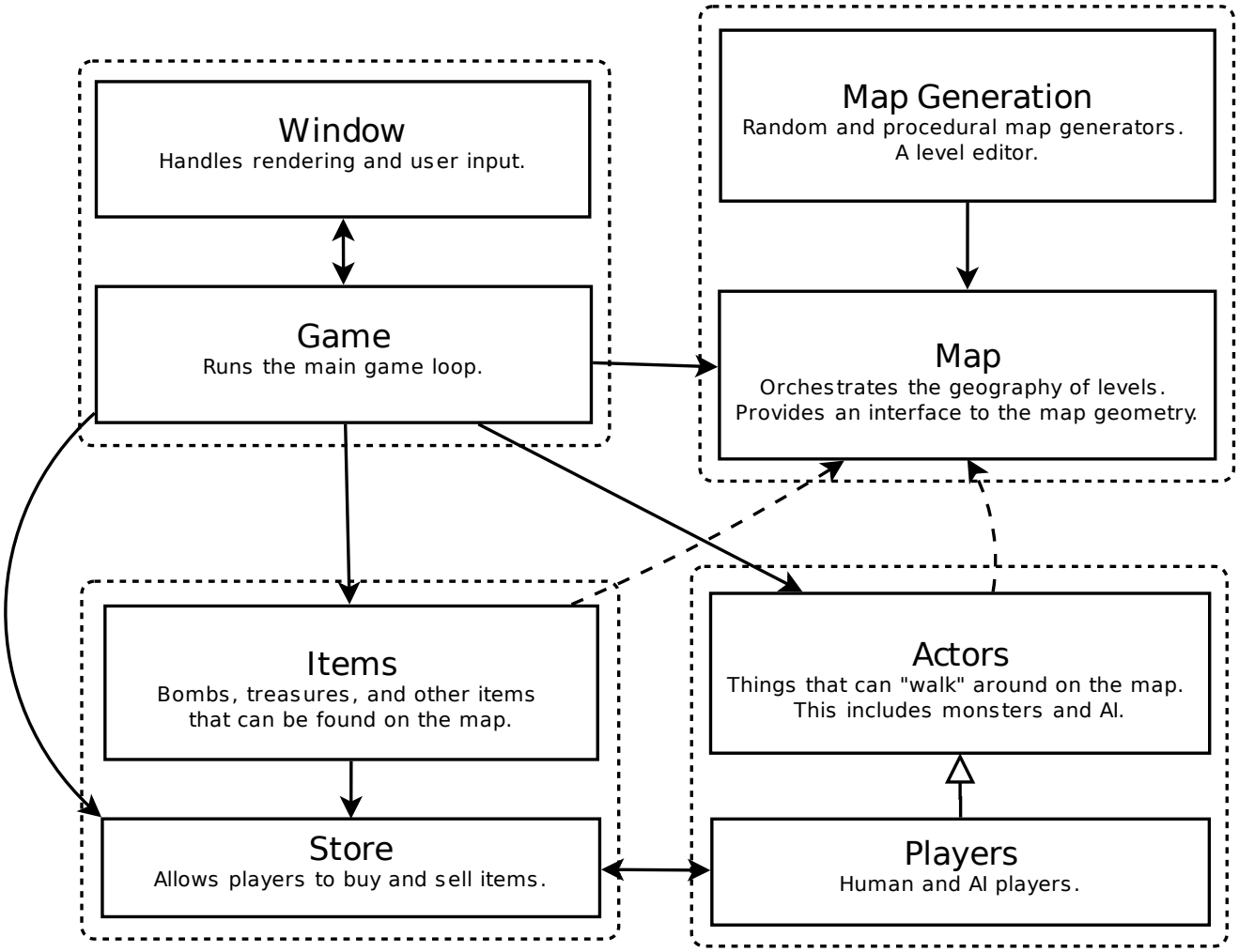
We discussed the possibility of netplay or some other type of multiplayer mode,

but decided that such features should only be considered once a solid foundation is in place. We strongly believe that a well-built set of basic features makes for good extensibility, including netplay and other large-scale improvements.

The features outlined above are the ones that have been discussed so far, and they form our starting plan. Plenty of work will be needed to refine them to excellence, though, and experience may bring to light unforeseen difficulties or new opportunities for exciting features. We recognize this and acknowledge that the specification will evolve along with the implementation.

Program architecture

The diagram below describes the general architecture of the program. The structure is pretty straightforward - the main class Game maintains the state of the game, Actors represent the moving and sentinent things in the game, Map describes the maps games are played in, Items are items to be collected and used and so forth.



The architectural design divides the the program into four quite distinct parts - the map and its generation, the actors, weapons and other items, and the overall game logic. This makes implementaion on separate parts of the program easy, and on the other hand divides logically separate things into their own sets.

In the design phase most of the problems we faced and discussed revolved around where the data is stored and how it's accessed, specifically around the way that objects in the game can request information about the map's geography — for example, when an AI player wants to know what's in the block in front of him, should that information be retrieved from the map or the block he is currently at? Should the map contain, for example, quick access to player locations or should it just find them once somebody requests them? We ended up storing most of the data for players and such in the Map object, so that map blocks don't need to contain references to other blocks or objects. Say, for instance, that an actor wants to know what the block he's facing contains. In order to get the location of that block, he would call a member function of the Map object, supplying his own location and the direction he's facing as parameters. Other possible queries include "first blocked (non-walkable) block in the given direction", "the set of blocks within distance X of the given block", and so on.

About maps and map generators

The maps will be made of blocks, which will be represented by their own class. The blocks will contain the relevant information regarding the location, such as the rendered image, terrain type and whether it has a player or a bomb in it.

Currently it feels like that there is no need to implement different map classes for single and multi player, they should work just fine on the same maps. If we end up implementing the procedural or "on the go" generated maps it seems likely that the basic map structure should be extended to handle the growing map, even though the interface will remain the same (for example some coordinates may be shuffled around when the procedural map is extended and the map has to update the corresponding elements, but the locations and elements residing within the map should be accessed with the same methods as in the case of the static map). The map public interface will contain all the relevant methods to access all interesting data about the map - the terrain types, player locations and everything else.

There will be two kinds of map generators, random generators and an editor that the player can use to create own maps. The editor will naturally have its own GUI. The possible procedural map generator will inherit the random generator, since they should share a good amount of code. The map generation and handling will also include a class for handling map I/O that is responsible for handling things like reading pregenerated maps from files and writing user generated maps into files. The maps will be stored in simple text files for easy reading and prototyping in the early stages of the project.

The maps should be covered with the classes drafted above. Next a few words about the map generation.

The Interweb is full of proposed algorithms on dungeon/map generation (see, for example, http://pcg.wikidot.com/pcg-algorithm:dungeon-generation for a list of links). The algorithm that we will initially implement will follow the general idea of all those found online:

1. Fill the map with solid blocks
2. Create a single room on the map
3. Pick a wall from any room
4. Randomly decide a new feture to add
5. Try adding the feature
6. If adding failed, go to step 3
7. Otherwise, add feature
8. Go to step 3 if the stoppping condition has not been met
9. Throw in some items and terrain features according to some heuristics or randomly

This basic set of random things should not be too hard to implement and extend once more sweet features are needed. There's a plenty of material, as stated earlier. Adding the procedural generation should be nothing more than to continue the generation from step 3 and adding more area to the map.

About items, store and inventory

Items will be implemented as an inherited class hierarchy. The items themselves are considered as "already placed" or used items, such as a ticking bomb, lootable treasures, teleporters etc. It will be easier to handle the logic behind these items this way. There will be a mapping from an item identifier(a string perhaps) to an amount, describing the player's inventory. The items work with the map to, for instance, create explosions or shoot bullets and track them. Most item collisions will also cause an effect, such as a player's gold amount increasing or hit points decreasing (e.g. from a trap).

The store will be an independent part of the game, which will be invoked between rounds in a multi player game and between levels of the single player campaign. In the store the player can spend their gold from treasures on items such as bombs, weapons, health packs and so on much like in the original game. The store will change the state of the accessing player's inventory. The store's user interface will be more sophisticated than in the original game, but serves the same purpose. On a hot seat multi player game, the players take turns to purchase items.

The inventory will be a simple structure holding the players current gold and item quantities in a dictionary of string keys and integer values. The actual implementations of the items will be inherited from the Item base class and they

will be added to an updater loop of all items within the game loop. Items serve a significant role in the game and define the major game strategies and mechanics. The pricing of items should make the player face difficult decisions in the store and form the main balancing of the game.

About Actors, monsters and AI

The Actor class is a superclass for creatures that walk around on the map. AI creatures (monsters) and Player characters both inherit it. Actor implements the logic of moving from place to place on each update, so that subclasses only need to worry about implementing updates for those characteristics that are peculiar to them specifically.

The AI in the game is predominantly concerned with the control of nonhuman Actors. The "interesting" bits of it are specific to each monster type, but there are also some tasks that are common to most AIs, such as pathfinding. Function object singletons will provide these facilities, such as the A* algorithm. Common facilities can have optional parameters, so that an AI that wants to use a pathfinding algorithm can submit its own cost function for map blocks, for example.

AI decisions can be understood as a series of "thoughts", each of which involves one or more decisions that dictate what the given Actor will do next. The goal of all AI thoughts is to make a plan that effectively brings the actor closer to completing some objective, such as (in the case of a simple, "player-seeking missile" type monster) to enter the map block that the player is currently in.

Thoughts can be computationally intensive, and if every AI monster tries to have a thought on every update, the framerate will plummet. To combat this, AIs will pace themselves: after having a thought and making a plan on what to do next (e.g., trace some given path generated by a random walk), an AI will set a timer that gets decremented by dt on each turn. When the timer passes zero, the AI will have another thought. One of the benefits of this is that monsters close to the player can adjust their thought frequency in order to react quicker, while those that are far away can safely go for many seconds between thoughts.

The Player class also inherits Actor. It is different from monster Actors in that it persists between game rounds and carries its state (inventory) over from one to the next.

The game loop and object management

We will use a general Game object to store the game state. The object will at first handle loading resources and initializing the required components for a game session. The object will then present the current scene (main menu, actual game play) to the user, react to user inputs within a game loop and finally take care of releasing any resources.

We will use a straightforward game loop that is divided into three major parts:

1. Event loop
2. Object updates
3. Drawing

The event loop will pump events from SFML and handle them. This mostly includes reacting to user input and window events such as a sudden exit event of the window. The events are pumped at the beginning of the frame to keep the object states, drawing and input all in sync. In the object update phase, critical game logic is handled. This includes changing the state of the game by updating each game object individually in a specific order and handling the most outward layer of game rules, such as a win/lose condition. The drawing phase draws the objects of the current frame onto the window and thus completes one full game loop cycle. The loop will add some padding time (pause) to synchronise with a pre-defined framerate if the game is intended to run in constant frame intervals. A general use 'dt' variable will be used such that any motion on the screen will be as fluent as possible, while maintaining constant simulation across different systems running the game so that the game experience stays the same.

Task sharing

Areas of responsibility are divided among the developers, roughly speaking, into the ones outlined by the architectural diagram. In addition to these tasks, graphical resources need to be prepared. The division is as follows:

Miro	Maps and Generators
Henri	Items and Store
Joonas	Actors and AI
Roope	Game logic and graphics

Towards the end of the development process, some areas will probably prove more labor-intensive than others, and development tasks will overlap. The primary areas of responsibility will nevertheless continue to be the ones described above.

Efforts will be coordinated in weekly meetings and via IRC. To make things go smoothly, interfaces will be made as clear as possible. Maps provide access to different aspects of level geometry, Actors have member functions to govern their movements, Items can be picked up from the map, and the Game contains logic to make it all tick.

Testing

A test suite will be written and maintained alongside the program code,

consisting of a set of unit tests for the nontrivial aspects of each component, as well as some tests that involve the central interactions between components. These tests will be run for each new iteration to ensure that new code does not break existing functionality. In addition, game mechanics will be aggressively play-tested for enjoyability — fun and games are serious business.

Schedule

Development will start with the production of a rough-but-functional prototype, which will then be iterated on to arrive at the finished game. We expect to produce this first prototype within about a week. From then on, an agile-like process will be used: weekly meetings will set targets for each week, progressing from essential components towards nice-to-have features and optional extras. The following table provides a rough roadmap for the project. The schedule is tight, but good results require hard work.

01.11 - 08.11	First working prototype (some maps, items, weapons and moving players)
09.11 - 15.11	random map generator, monsters with AI, item store, more weapon types
16.11 - 22.11	Minimum requirements fulfilled. Work started on some optional extras
23.11 - 29.11	Sweet graphics and audio
30.11 - 08.11	Bug fixes and finalization, documentation

External libraries

SFML will be used for graphics, input handling, and other tasks. At present, we foresee no need for any other external libraries.