Project plan

The scope of the project

The objective of this project is to design and create a 2D top-down perspective “minebombers”-like game implemented with the C++ programming language. We plan to realize our own custom game engine that supports all the needed features of our game. External libraries are used to help with the implementation of certain parts of the game like graphics, sound and resources.

Game Engine

The main emphasis of the implementation will be in the game engine. A game engine should be generic and flexible to allow fast and easy iteration of the game design. It should be designed in a way that facilitates the addition of new features with ease, as the game needs them. The game engine should be kept separate from the game implementation as much as possible to maintain a good code structure.

Some features that the game engine will support is 2D rendering, sound and music, resource loading and management, input event handling, and map generation. Stretch goals for the engine are support for network multiplayer and more efficient resource loading using multithreading.

The game engine should, for the most part, be well separated from the external libraries and dependencies by using interfaces as a contact surface of the external libraries.

The quality of the engine will be evaluated with unit tests. As the testing of a game project can be challenging, we will keep unit testing concise by limiting testing to core parts of the game engine like the algorithm implementations and collections that are developed for the engine.

Game

The game will be played from a 2D top-down perspective; the world will be generated in a rough 2D grid, and most game logic will be restricted to the grid. The game will be designed to be played locally by multiple people. Players will be able destroy parts of the terrain to modify the game world. This will give players a possibility to make passages and rooms that will help them achieve their goal.
The game world will be procedurally generated. Some possible algorithms for maze generating are depth-first and Kruskal's Algorithm, *See Appendix B and C for more information*. Generation will create a world with multiple terrain types that will have various effects on the gameplay.

The game design will be refined as the development moves forward using an iterative design process.

Major architectural decisions

Core engine architecture

See Appendix A for a diagram of the core engine architecture.

Scene Graph

The very core of the game engine consists of a scene graph. The scene graph consists of nodes, and nodes represent all the “physical” aspects of the game. Nodes all contain a transform object, which represents where in space the node exists (relative to the parent). A node can have various attachments, such as a sprite, an animation, or a camera. These inform the game engine what the scene looks like visually. On top of that, a scene can have *behaviors*. These inform how the node should update each frame, as well as how the node reacts to events.

Messaging

Nodes should try to update themselves as much as possible based on their surroundings (which they can do by iterating the node-tree). Sometimes, however, it can be necessary to communicate between nodes: this is achieved with a messaging system.

The messaging system has two main components: a message publisher, via which all messages are sent, and message subscribers, which receive the messages. Messages are composed of two elements: an address and an event.

The address has the following syntax:

```
[socket:][path]
```

The socket tells which message listener should receive the message. The following part (path) tells who, within the socket, should the event be passed to. This can be of several levels, using “/” for level divisions, eg: “gameWorld:vehicle1/wheel3/bolt4” would send the message to bolt4 of wheel3 of vehicle1. The message listener takes care of parsing the path field and returning the valid event handler.

Messages can be sent directly to a socket too, to be handled, without an additional recipient, ie. “gameWorld”. In this case the message listener directly handles the event.

The event itself typically contains an action type, and relevant variables. For example, an audio message might look as follows:

```
address: “audioPlayer:grenadeExplosion”,
event: {
      actionType: play,
      loop: false
}
```

This would send a message to the audioPlayer, to the grenadeExplosion audio clip.

Resource management

All game assets and configurations will be kept and loaded from separate non-code files. This includes images, sounds and other media, but also level, weapon and other configuration files. The game logic behind each individual parameter obviously has to be coded, but no configuration should be created in the code. This makes it easy to create additional content or mod the game. The separation of the configuration files also makes tweaking and testing different parameters easier. This will make iterative game design development faster because the engine doesn’t have to compiled every time a configuration is changed.

Preliminary schedule

7.11	Got project topic
8.11 - 14.11	Planning, game design and game engine development
14.11	Project plan deadline
14.11 - 24.11	Game development. First milestone ● Basic game is working ● The game engine implemented and working ● Have get a good idea about the final result and game design and a concrete direction for the second milestone
24.11 - 30.11	Mid meeting. End of the first development milestone development
24.11 - 12.12	Game development. Second milestone ● Game design implementation ● Final design iterations
12.-16.12	Final demo. End of the second development milestone development.

Design rationale

Game Engine

The game engine is designed to keep the hard work behind the scenes. When implementing gameplay, adding behaviours and features should be as simple as possible: this is realized by keeping a simple and rigid structure (scene graph), and limiting communications between game objects to messages.

By abstracting the “engine” from the “game”, iterating on the game design becomes easier.

Development practices

Git

Branches

Our project will follow a simple branching structure. There are two main branches: master and develop. Master contains “releases” (somewhat stable versions of our game), whilst “develop” is the branch in active development. New features are created in branches originating from develop, and merged back to develop. Hotfixes are branched from master to master, and applied to develop as well.

Merge Requests

Merge requests (when a feature has been completed) are performed when adding a feature branch to the develop branch. To ensure the code is of a good quality, and that errors and bugs are minimized, a second person in our development team will perform a code review before merging.

Commits are automatically built and tested: commits that fail will not be merged.

Code Style

The code style has a heavy emphasis on features introduced in C++11 and C++14. All variables (where applicable) are declared with the ‘auto’ keyword. Functions are declared with auto syntax as well:

```
auto func(params) -> returnType
```

This keeps the style consistent and easy to read. A strong emphasis is kept on the legibility of the code: this way documentation can focus on explaining what a realization does without needing to explain how.

Testing

Every time a new feature is added, it should also be sufficiently unit-tested. This is done using the Catch frame. Sufficient testing means bugs are caught early-on, and unexpected behaviors can be mitigated later on in the development cycle. This also means internal implementations are easier to change: the tests help inform if the modifications were implemented correctly.

Project Management

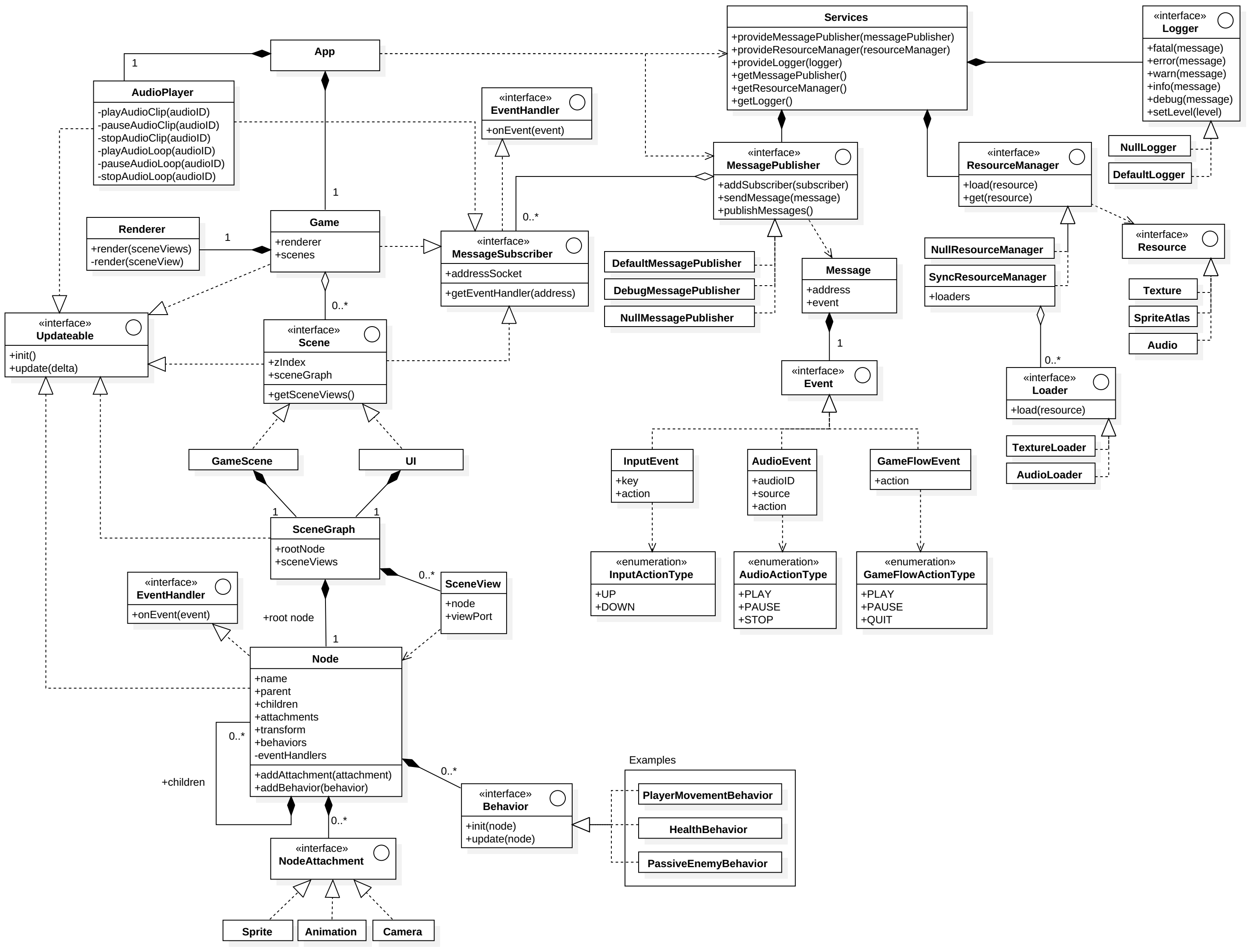
For managing the project, we have a slack group for communication, which includes gitlab webhooks that keeps everyone up-to-date about changes made in the code-base (as well as whether test pass or not). On top of that, we have a wiki for documenting aspects such as commit practices, and we have a Kanban-style task board for managing what needs to be done.

Tools and libraries

Build process:	CMake
Communication:	Slack
Generated documentation:	Doxygen
Automated testing:	Gitlab CI
Continuous Integration:	Gitlab CI
Task management:	Gitlab task board
Testing	Catch framework
Graphics:	SFML
Scripting/Resources	Sol2 lua library

Appendix A

Model::Main



Appendix B

[https://en.wikipedia.org/wiki/Maze_generation_algorithm]

Appendix C

[http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm]