**Faculty of Science**
Bachelor: Computing Science

Academic year 2014-2015
Date: Friday, 26 June 2015

# Dunwich: Behind the Dungeon Crawler

## A report on the app-building process

Tim van Dijk, Luuk van Blitterswijk,
Ward Theunisse, Martijn Heitkönig

Radboud Universiteit

## Table of contents



## Introduction

This report consists of a review on the app-building process for the android game Dunwich. Dunwich is a rogue-like Dungeon Crawler based on the works of H.P. Lovecraft and is the result of the teamwork of a group creative minds during the course Research and Development at the Radboud University Nijmegen.

The goal of this report is therefore to provide documentation on the description of the app itself and the design process of the app, and also to provide a reflection on the app-building process and the final state of the app. This is done by dividing this report into 3 sections.

The first section provides a description of the app. In particular, a global description of the app, as well as the specifications of this app, and a justification of the choice for this particular application is given.

The second section proceeds with the design of the application. This section elaborates on the global design of the application and the detailed design, in terms of essential classes and methods, and describes the justification behind the different design decisions.

The third section is the final section of this report. It consists of a reflection on the app and the app-building process, from which a conclusion is drawn with respect to the achieved results and the implications for future projects.

## Description of the application


### *Global description*

Dunwich is a game in which the goal is to descend through various randomly generated floors, retrieving an artefact, and bringing it back to the start. The most important aspects of our game are of course the basic elements of movement, combat, and pickups. However, we wanted to differentiate ourselves from other similar games by the means of a lighting system, and randomly generated corridors. We tried to put various elements from a genre in our app that is typically aimed at PC users, which meant that we had to translate a lot of actions that are usually performed using a keyboard, to various types of input on a touchscreen.

*Movement*

Movement is done through the means of an overlay, in which the screen is divided into different segments. When the user taps the screen an overlay pops up with the current segment highlighted, as seen in the screenshot. If the user made a mistake in the direction he would like to go, he can correct this by dragging his finger to a different segment. The segment in which the user releases his finger will be registered as the direction in which to move (or, in case the middle segment is tapped, to perform an action on the player). If a player moves into the tile of a door, he kicks open the door. If a player moves into the tile of a monster, he kicks the monster. The default action is using your feet against whatever it is you're moving towards.

*Combat and input*

Combat is partially done by walking into something, this performs a “kick” action by default but that isn’t really effective against monsters. Players can therefore use weapons too: Currently Dunwich features two types of guns: a pistol and a shotgun, and two types of melee weapons: a labrys, and a dagger. Using a gun requires a different kind of input than using a dagger. You can’t really “aim” a dagger at a non-surrounding tile; their range is only 1 tile so the segment wheel used for walking is good for this purpose.

Shooting however, requires you to aim at a certain tile, so when a gun is selected, the regular input wheel changes to a crosshair, which can point to each individual tile that is currently visible to the player. When the finger is released from the tile, the bullet flies in the direction of the tile, and keeps moving along that angle until it hits a wall. Lanterns don’t require any aiming at all, you can either turn them on or off, therefore a simple tap on the screen suffices. The same is true for our clairvoyance spell. Tapping the screen with clairvoyance selected shows the path to either the relic, or, in case the player is in possession of the relic already, the exit.

***Justification of the chosen application***

The decision to build a rogue-like dungeon crawler for android was made, because the market of rogue-like dungeon crawlers for android is currently mostly served by two different variants of rogue-like dungeon crawlers, namely StoneSoup and Pixel Dungeon. The latter is open source, which resulted in different variants of this game on the play store. Thus, although it looks like that there are quite a few other rogue-like dungeon crawlers, they are basically the same game.

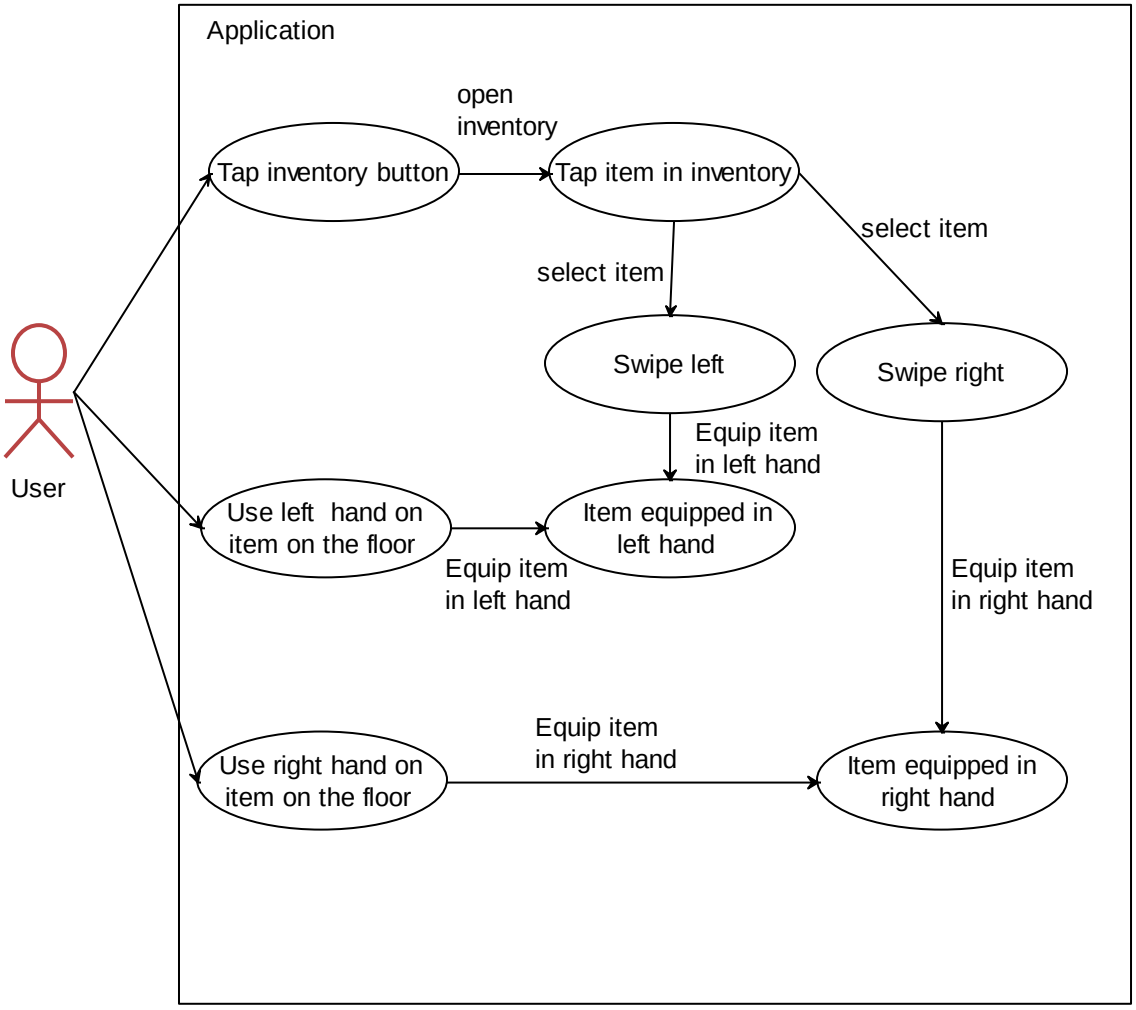
The fact that the competition consists of just two variants of rogue-like dungeon crawlers means that there is a gap in the market that can be exploited. The expectation is that, while current users are being saturated by the competition and with the growing market for rogue-like dungeon crawlers, a relatively high market share can be gained with prospects for a future profit, if our app can differentiate itself enough from the current competition.

This differentiation has been achieved by implementing more user-friendly navigation controls, the possibility for free-roaming in the over-world instead of just running through dungeons, a lightning system, and at last a sanity system, which is to be implemented in the near future.

***Specifications***

It's kind of hard to make a use case model for this app. Without going into detail on the game, all the player can do is start a new game, die, and win. Dying and winning triggers a restart.

The use case diagram of our game would look a bit boring (the user starts a game, and then either dies, returns without the relic, or returns with the relic. In any case, the game starts over). So we decided to make a use case diagram for equipping an item.



## Description of the Design

*Model*

The game features seven randomly generated levels, and two hardcoded ones. A level is a 2d array of tiles. Tiles contain a resource, and can be either a floor or a wall. Tiles also contain a list of all the world objects that are present on that tile. World objects are objects in the world, in the broad meaning of the word. There are two types of world objects: actors and props. The difference between the two is that actors can perform actions, hence the name. Whenever the level advances a step, all the actors are asked to perform an action.

*Levels*

A level contains two lists of world objects; a list of all the actors, and a list of all the props. Although actors are also saved in the tiles, keeping a separate list is a lot more time-efficient. To make all actors perform an action, you don't have to check the entire list of tiles for the presence of an actor. This also allows us to save objects outside of the level. In case you try to retrieve a tile outside of the level, a standard wall tile without contents is returned. The fact that objects are saved in tiles at all has multiple reasons; firstly, when an actor performs an action that has something to do with nearby tiles, you don't have to search through the entire list of actors and props in the level. Instead, the individual nearby tiles can be checked, to find out which are affected. Wall tiles are allowed to hold objects, because we support ghosts that move through walls, and in the future we would like to add spells that allow you to walk through them.

*Props*

Props have multiple subclasses:

Firstly, there is a subclass for staircases. Staircases contain extra information on which level they are connected to, and to which coordinate on said level. They are not connected to a staircase in another level. This allows for one directional connections, like holes in the ground or trapdoors, and makes the process of placing staircases easier. This way you won't have to place a staircase without connections on both floors, and add in those connections later. Instead, you can place a staircase with a connection from the first level to the second level, and place a staircase on the second level with a connection to the first.

Secondly, there is a subclass of containers. These are props that can contain items, but still exists after being emptied out. Also, items can be stored in them. This makes it possible to place objects like cupboards, chests, and lootable corpses.

Thirdly, there is a subclass called itemprop. These are props that contain 1 item, and cease existing after it has been picked up. This allows for placing items on the ground that can be looted, or items that can be dropped from the inventory.

Furthermore, a tile keeps information on whether it is walkable, and if you can see through them. World objects can alter those properties; they have the ability to block movement (e.g. tables, closed doors, or, in a future version, boulders), and block sight (e.g. closed doors or big monsters). When the collection of world objects in a tile changes, these properties are rechecked.

*Objects*

Objects can contain a light source, which is either on or off. Whenever the level advances a step, a check is performed on all world objects, in order to find out whether they emit light. If they do, the level gets lighted from that light source. A Light source consists of a strength, a light intensity falloff, and a colour. For every light source a list is made with all the tiles within reach of the falloff. Next, for every tile within the radius, a line of sight check is performed through a raycast. Then, for every tile in that line of sight a lighting value is calculated according to the intensity and falloff. Then, the tile's light intensity is set to the maximum of the current intensity, and the new one. Upon recalculating the light value, the entire level loses its lighting, so that turning off a lantern for example, will make the surrounding tiles dark again. After the calculations for all light sources are complete, a fading method is applied to all lighting, so that walls are lighted indirectly by the floor, and the lighting on the border gets smoothed out.

*Actors*

Actors are made up of different subclasses: Players, monsters, effects, and chat triggers.
Effects are actors without intelligent controls. They manage an internal clock of how many turns they exist. Example effects are laser beams, toxic gas, bullet traces, magic projectiles, or simply visual effects.

*Chat triggers*

Chat triggers are objects that cannot move, but trigger the log when a player steps on it, after which they disappear. They can either be invisible, or contain an image. An example chat trigger with an image would be a question mark on the floor, which gives the user an explanation on various things. An example invisible chat trigger would be a vivid description of a room with a horrifying monster.

*Monsters*

Monsters are actors with HP, each featuring their own AI. They are able to detect whether the player is in their field of view, and have access to pathfinding. Monster specific acting is part of their AI. Take the mimic for example: A mimic looks like a door and doesn't move at all, until a player stands next to it for two turns. This triggers the mimic to change into his monster form, giving it a different texture. In this form it tries to chase and damage the player, given that it is still in its field of view. When the mimic loses the player, it tries to find the closest location in which a door could spawn, walks to it, and changes back into its door form.

*Player*

The player is an actor with HP, controlled by the user. It contains a list with inventory items, and an array with a size of two, which resemble the player's hands. The player also contains his next action. As opposed to monsters, the player doesn't think of what to do every turn. Instead, it gets this information from controller input, that has been filtered by the level.

*Items*

The class item contains a list of all allowed items and their sprites. There is an optional sprite list for items which are stored in the players hand and are selected, or are lying on the ground. A book for example, will look like it has been opened, when selected. A can of beans on the other hand will look the same whether it has been selected or not. A gun is positioned diagonally on the ground, but looks straight when stored in the inventory. The class item also contains a list of control configurations linked to items. This is used by the controller to, depending on the item used, offer a different control set. Items are allowed to lack a control configuration, but then they can't be used to interact with other things.

*Interactions*

There are two types of actions: actions where an item is used, and actions where an item is used on a tile. An empty hand or empty selection is seen as an item. This doesn't mean however that they are legal item objects. An empty selection is translated into an action for which the player's feet are used. The class Actions keeps a huge list of available actions for both kinds. The tile for item-tile-interactions is obtained from the controller, through the selector type. This means that you can choose your surrounding and current tile from the radial input, and one tile from the 11x11 tiles around you from the crosshair input. A list of interactions is then retrieved from the selected tiles. Indirectly by checking if the selected tile is a wall, monster, or air, directly by adding all the world objects present in that tile. Next, all the items on the action list are matched against a combination of the used item and the interaction goal from the prepared list. From all results, the action with the highest priority is chosen. This action is then set as the player's next action. This method is chosen to keep the interactions intuitive and correct, without the need of submenus. For example: the player stands next to a monster. This monster is in a doorway and stands on top of an item. The player uses his hand with on the tile which hosts the monster. The found interactions are: picking up the item, closing the door, and hitting the monster. Hitting the monster has the highest priority, so that action gets performed. Walking is implemented the same way. Walking is just an interaction between your feet and the air. A player will never walk through monsters, walls or props, because the interaction of "walking" either doesn't exist, or doesn't have the highest priority.

*Event log*

The game keeps a list of all events that occurred. Most events in the game give an output through the event log. The most recent events are visible from the main screen, older events are readable by tapping the event log and scrolling through the list. An example event would be "*You loot the chest and add the bullets, can of beans and book of Belial to your inventory.*" or "This door won't budge". Walking on the question mark in the beginning of the game triggers the tutorial, which gets written to the event log.

*View*

The view has a set of Booleans which keep track of the current game screen. These Booleans are managed by the controller. The controller gets information on whether the screen has to be redrawn, and which buttons or polygons are selected. When the model changes, the view gets updated via an observer-observable relation. The view itself keeps a local copy of the visible section of the current level, focussed on the player. Every time the view gets updated by the model, this copy gets refreshed and a new copy of the inventory is made. When the level gets redrawn, the local copy of the section of the map is used, and a set of all the different resources that are used by the tiles and objects in that part is made. For every resource a bitmap is created. This is to prevent duplicate bitmap generation, because this is a resource intensive operation. Next, for all the tiles the bitmap is drawn at the correct scale, via a colour filter of the lighting of that tile. Then the world objects are drawn the same way, in the order of their drawing rank. If the dimensions or orientation of the screen changes, all elements are rebuilt and passed to the controller. Depending on the currently specified screen, other UI elements are shown.

*Controller*

The controller itself has an internal set of Booleans which indicate in which screen the game is at the moment. It keeps a list of the items that are currently held by the player, because the input method depends on the item that the player has selected. The controller talks to the view and the model, and lets the game advance by one step when the user chooses an action that makes the game progress. The controller talks to the view in a way that is dependent on the current screen. The screen can be normal, full screen event log, or inventory. This way you can scroll through the event log, and swipe and scroll in the inventory. It gets dimensions, quadrilaterals, and polygons that are required for input from the View. This is necessary because the screen adjusts itself depending on screen dimensions and screen orientations.

*Level generation*

At the beginning of the game, 7 levels are generated in a way that is discussed in the justification of the design. Afterwards, the levels are then filled with monsters, torches, and items, influenced by level depth. Levels are connected through the means of stairs, the process of which was discussed in the subclass section of Props. A pair of staircases (one heading up, one heading down) are placed in a way that maximises the distance between the two.

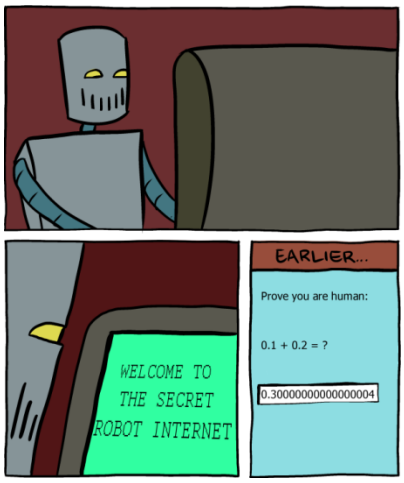
*Villa generation*

The first two levels take place in an abandoned villa, that is not randomly generated and serves as a preparation level. In here, the player learns the basics of the game, and it's possible to find items that can help you in the other levels. The villa is entirely generated by a different class, with methods that place hardcoded rooms and pick only a few elements at random, like the amount of food the player can find in the kitchen. Our original plan was to save this level on the disk, but we decided against it, because it would take up a lot of development time that we couldn't afford, and would be of better use someplace else.

## Reflection and Conclusion

Two problems that we solved were pathfinding for monsters, and an algorithm to generate (pseudo) random floors. The way we tackled the pathfinding problem was by using the A* search algorithm. We would have rather used the JPS (Jump Point Search) algorithm, because it is an optimised version of A* and can be used under the condition that a diagonal movement costs as much as movement in the cardinal directions. The reason we settled for A* search was that we were already familiar with it, and were in a bit of a hurry. At the moment it doesn't really matter and the game runs fairly fluently as it is at the moment, but when facing, say, fifty enemies, the game becomes unplayable. In the future we would like to implement JPS as an optimization technique, because it still matters on slow devices.

Since we had our code for A* search already, we decided to reuse this part of the code in our FoV (Field of View) check. In summary, if the length of the shortest path (which could run through a wall) between A and B is exactly the same as the path between A and B that we actually found, they are in each other's FoV. A problem that occurred here, was the finite precision in doubles. Small differences (e.g., 18.000000000000000000000003 as opposed to 18) resulted in unexpected behaviour, which was fixed by allowing a small margin when checking if the two were equal.




*http://www.smbc-comics.com/?id=2999*

The second problem was level generation. It is an essential part in our game, because playing exactly the same game would get boring fast. Tim had some experience with level generation, as part of a hobby project. This code worked most of the time, but not always.

The initial method of creating a floor was to place chambers at random locations on the floor, and trying to dig out paths between them, by applying a path finding algorithm from the centre of each room and digging out parts of the wall to connect them to other rooms. For a demo this was fine, but for the final game every chamber was guaranteed to be reachable. If the staircase would end up in an unreachable room, the player wouldn't be able to progress, which would be devastating.

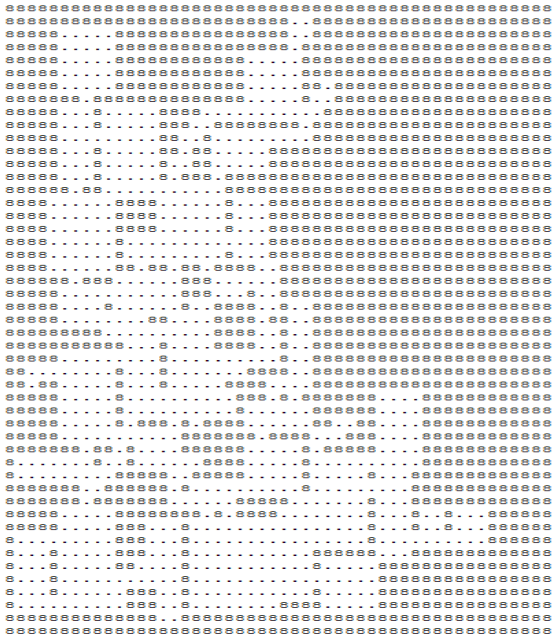
Our new process of level generation is:

1. Place a room at a random spot in the level
2. Choose a wall
3. Choose which chamber or corridor you would like to add to the level
4. Try to place the new chamber or corridor on the other side of the chosen wall
5. If it succeeded, remove the chosen wall and connect the current level with the new chamber or corridor
6. If there aren't enough chambers or corridors, go back to step 2.

It works, but it's not as good as we hoped it would be. An example level resulting from this algorithm would be:

It works, but not as good as we hoped it would. For every corridor, at least one of the ends is connected to a corner. Also, this implementation did not allow for the use of custom, hardcoded rooms. We wanted to use those for special rooms, like boss or event rooms.

To solve the first problem, we tried to move the chambers before placing them. This worked pretty good, except sometimes the rooms wouldn't connect to other rooms. In hindsight, this was due to an error where the chambers would get moved in the wrong direction.



Then we tackled the problem of custom rooms. The difficult part was to make sure the walls of a custom room would not get overridden by floors. This meant rewriting the entire level generation code so that null tiles became expandable walls (in the context of step 2.), whereas hardcoded walls didn't.

It was a bit of a hassle, but after a while the level generation code was complete. The only addition that was done at a later time was adding torches, items, doors, and monsters to the rooms, but that was fairly easy to do.

We do still have aspirations for the future:

- A large diversity in custom rooms, because as of now we've only got three proof-of-concept rooms.
- Different floor styles, because every floor is grey at the moment. Adding in things like a cave system, an organised looking cave, underwater cave, etc. would bring a lot of diversity in the game.
- A dynamic amount of rooms and chambers, which is dependent on the size of the level.

Our current level generation in pseudocode would look like:

1. Check which level we are generating. If it's the bottom level, go to 1.1. Else, go to 1.2.
   1.1 Create a relic room, and add it to the level
   1.2 Create a chamber, and add it to the level.
2. Look at the whole level, and choose a wall.
3. Choose whether to add a chamber or a corridor to the level
4. In case we add a chamber, go to 4.1. Else, go to 4.2
   4.1 Try to create a new chamber on the opposite side of the chosen wall, by sliding it along the wall and checking whether there is a position in which it fits.
   4.2 Try to create a new corridor on the other side of the chosen wall
5. If step 4 succeeded, add objects that belong in that chamber and remove the wall that separates the new chamber or corridor.
6. As long as there are less than 50 chambers or corridors added, go back to step 2.
7. Add torches to random wall tiles
8. Replace all null tiles with walls
9. Spawn monsters
10. Spawn items
11. Spawn doors

**Current state of the game**

To start off, we don't think that our app is ready to be released to the public yet. However, we are of the opinion that making this app was worth the time. We each learned new things about programming, art styles, and version control, which is still valuable to us. There are other games in the Play Store at this moment that do something like our app. However, we tried to differentiate ourselves through the means of a sanity system, which – as far as we know - would be unique. We didn't have enough time to implement this feature however, so as of now we don't really offer anything advantageous over our competitors.

At this point, we don't really think our app would be worthy of a Play Store release in terms of gameplay and fun. We did however build the fundaments of such a game, and we would like to expand it into something that is actually fun and rewarding to play. The way we would like to do this, is to add more content in various forms. This would include new items, weapons, monsters, AIs, and a boss level. To make the app more visually appealing we would like to implement coloured light sources. Our codebase supports this already, but we don't utilise it yet. This would make light sources like the lantern add a red glow to the surrounding tiles and walls, whereas a magic light source could emit more of a blue coloured light, which contributes to a more dynamic looking game.

In conclusion, we're happy with the progress that we've made up to this point. Nobody had experience with developing applications for android in the past, yet we were able to sketch out a plan and create a working game from scratch in about five weeks. Granted, there are a lot of features that we would have liked to implement, but were unable to. We had high aspirations when we started, and couldn't really estimate how much time this whole project would cost, so we were a little optimistic on how much we could actually do in that timeframe.

We're looking forward on continuing the development of Dunwich, and hopefully one day releasing it on the Play Store!