Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Procedural Generation in the mobile physics puzzler Gravel

by

Björn Jansson

LIU-IDA/LITH-EX-G--14/060--SE

2014-06-16
Final Thesis

Procedural Generation in Gravel

by

Björn Jansson

LIU-IDA/LITH-EX-G--14/060--SE

2014-06-16

Supervisor: Erik Berglund
Examiner: Anders Fröberg
## Contents

Abstract .................................................................................................................. 1

INTRODUCTION ....................................................................................................... 1
  Motivation ............................................................................................................. 1
  Research Question ............................................................................................... 1

BACKGROUND ......................................................................................................... 1
  Thesis Work ......................................................................................................... 1
  Gravel Gameplay .................................................................................................. 2

THEORY ....................................................................................................................... 2
  Procedural Generation ......................................................................................... 2
    Different Distinctions for Procedural Generation .............................................. 2
    Online or Offline ............................................................................................... 2
    Necessary or Optional Content ......................................................................... 2
    Random Seed or Parameters Vectors .............................................................. 2
    Stochastic or Deterministic Generation ........................................................... 3
    Constructive or Generate-and-Test .................................................................. 3

METHOD ...................................................................................................................... 3
  Implementation ..................................................................................................... 3
    Development Tools and Programming Language ............................................ 3
  Agile ...................................................................................................................... 3
  Figure 1. Example of generation by the first implementation ............................ 4

RESULTS .................................................................................................................... 4
  First Implementation ............................................................................................ 4
  Second Implementation ......................................................................................... 5
  Figure 2. Example of generation from the second implementation ................... 5
  Third Implementation ......................................................................................... 6

DISCUSSION ............................................................................................................... 6
  Results .................................................................................................................... 6
  First Implementation ............................................................................................ 6
  Second Implementation ......................................................................................... 7
  Figure 3. Code example of the cluster generation function ............................... 8
  Third Implementation ............................................................................................ 8
  Method ................................................................................................................... 9
  Agile ....................................................................................................................... 9
  Implementation .................................................................................................... 9

CONCLUSIONS ......................................................................................................... 9

REFERENCES .......................................................................................................... 10
Abstract

This thesis is concerned with procedural generation in a genre of games where it is not very common; 2D physics-based puzzle games. I will explore which different methods of procedural generation will work with this kind of game, if any. If there is an easy way to implement procedural generation it could cut down on the game designers work load significantly, which would allow more content to be produced at a higher pace.

INTRODUCTION

Procedural generation allow video games designers to reduce the time they spend on content creation for a game, which means more resources can be spent on other parts of the game. If implemented the right way, it could also allow new content to be continuously generated, which means games would stay fresh for longer.

The purpose of this thesis is to explore if there is any way to implement and take advantage of procedural generated content in the game Gravel, which is a 2D physics-based puzzle game for Android.

Motivation

There are many games that use procedural generation to generate maps and levels. Civilization 5 and Age of Wonders 3 uses it for map generation. The Borderlands series uses it for weapon generation. Diablo 3 uses it for both weapon and dungeon generation. However, there are not as many examples of procedural generation when it comes to physics-based puzzle games.

If you want to generate a complete level for a puzzle game you need to be able to check if the level is solvable and also if the level is challenging. This can be relatively easy for a puzzle that exists in a space that can be represented by an expression tree and/or can easily be evaluated by a simulation-based function. [1]

A game that uses a grid system can easily be expressed with an expression tree. It’s possible to describe a castle as “this room leads to two corridors, the first corridor leads to another room and the second corridor leads to two rooms” and so on. However, Gravel can’t be described in this way. The game is set in space and there are no barriers. The elements aren’t connected to each other in a way that can be expressed with an expression tree. This will make it difficult to use a simulation-based evaluation function to see if the level if valid. One would basically have to run the game to test the level, since it is hard to test in a more abstract way. It would thus make sense to evaluate whether there is a way to work around the problem or if there some other ways to make use of procedural generation in a physics-based puzzle game like Gravel.

Research Question

- How can you implement procedural generation in a 2D physics-based puzzle game?

In what way can you implement procedural generation in a 2D physics based puzzle game so that it actually contributes to the game? If you can’t create a whole level can it still be used to create parts of a level?

BACKGROUND

Thesis Work

The task that I was given for my thesis was to implement procedural generation in the mobile game Gravel. Gravel is a 2D physics-based puzzle game that is under development for Android. The game was already partially done, so my job was to see if there was any way the game could benefit from procedural generation.

The game had been developed in Gideros Mobile, so that was what I would use for my thesis work too. Gideros Mobile is a cross-platform game development tool. It enables the user to create a game on their platform, and then export it to iOS and Android.

There were two other thesis workers who worked on the game alongside me. One of them worked on Facebook integration and the other worked on improving the UI. Even though we had different thesis questions we worked together on fixing bugs and improving aspects of the game.

1 http://www.firaxis.com/?/blog/single/procedural-terrain-generation-in-sid-meiers-civilization-v
2 http://ageofwonders.com/dev-journal-visualising-the-battlefields/
4 Games with miscellaneous_procedural_effects
5 http://us.battle.net/d3/en/forum/topic/6147286293
6 More information on Gideros Mobile can be found at http://giderosmobile.com/
Gravel Gameplay
In order to understand how the level generation works it is necessary to understand how the game functions. In order to help the reader understand the game, I will explain the core concepts and explain how some things changed during development.

The game is set in space and the player controls a spaceship which he tries to get into the goal area. However, the spaceship doesn’t have a built in propulsion system, instead the player needs to use gravitation nodes and repulsion nodes to fling the spaceship into the goal. The number of nodes that the player starts with varies between levels and is set in the level editor. These nodes can be increased by picking up power ups during play.

Asteroids are placed around the map and they can be affected by the gravitation and repulsion nodes just like the player. The asteroids serve as obstacles for the player to get around. If an asteroid collides with the spaceship, the spaceship will lose health. After losing enough health the spaceship will explode and the game will be lost.

At the end of the project, a few things had changed with the game. The repulsion node was removed because it made removing obstacles easier than to try and get around them. The limit on how many nodes could be placed was removed instead the scoring system was changed so that it focuses on how many nodes that had been used. The power-ups just give extra points instead of giving extra nodes. Most of these changes were done pretty late in development and were implemented between the second and third implementation. They would have made a big difference in the first implementation, but they didn’t impact the second and third implementation very much.

THEORY
Procedural Generation
Procedural generation refers to the process of creating game content with algorithms. Instead of having a designer create a level or an artist make a texture, you can use an algorithm that takes random values and uses these to create it instead.

Procedural generation can be used for a many different things in a game, from generating a whole level, to just generating the loot you get from enemies. Procedural generation can also be used to generate more aesthetic game elements, like textures and trees. [1, 2, 9]

Different Distinctions for Procedural Generation
There are a few different distinctions that can be used to classify different types of procedural generation algorithms. These distinctions are not binary choices though, they are more like a spectrum on which any given procedural generation algorithm can be placed. These distinctions can be very useful for determining how to implement procedural generation into a given game. They help the developer think about their game and analyze what they wish to accomplish with procedural generation and how should implement it with those goals in mind.

Online or Offline
This distinction refers to when the generation occurs. Online generation refers to generation that is done on the spot. An example would be when you choose to play a game on a random map and the game generates that map to your specification. Offline however refers to when the generation is done by the developer. The developer then decides if the resulting maps are good enough, and can choose to scrap it entirely or edit the result. An example of this would be the trees in Oblivion, which were created with procedural generation[6]. [1, 2]

Necessary or Optional Content
The difference between necessary and optional content is as clear cut as the title would suggest. Necessary content refers to generated content that is necessary to complete the game, for example a dungeon that must be completed in order to win the game. Optional content on the other hand is content that isn’t necessary to complete the game. An optional dungeon that is not related to any quest would be a good example of optional content.

These two extremes affect how important it is to validate the result of the generated content. If the content is optional it doesn’t matter so much if you can’t get to the end of a dungeon or if the loot you pick up is useless. But if the content is vital to the game, one needs to be sure that the content can be completed and will not make the game unplayable, should something go wrong. [1]

Random Seed or Parameters Vectors
This distinction is concerned with how much input the user has over the generation. Random seed means that the only thing that is need for generating game content is a number or a vector to initialize the random function. An example of this is the world generation in Minecraft, which uses a string

of characters, the random seed, to create a new world. On the other end, the procedural generation function might take a number of parameters that decide how the content will be generated. For example how many rooms, floors, stairs and doors a house should have. The function then generates a solution with the help of the parameters that it has been given. [1, 2]

Stochastic or Deterministic Generation
This refers to how random the generation is. A stochastic generation means that the generation function doesn’t return the same result just because it got the same parameters. A deterministic generation on the other hand will do just that. The seed is not included as a parameter here. If it was, all generation would be deterministic since there is no way to generate truly random numbers in programming. [1]

To use an example generating a house with the help of parameters, a deterministic house generating function that gets told that the house should have two floors, eight rooms, elven doors and two stairs will always generate the same house. If the function was stochastic instead, the house might have variations between runs.

Constructive or Generate-and-Test
This distinction refers to how the validity of the generated result is controlled. A constructive algorithm checks the generated content during generation.

The generate-and-test method means that you generate content and then use some method to test it to see if the result is valid.

There many different ways to test the result, one of the most common is to use a search-based method. This means that the test function grades the candidate content instead of either accepting or rejecting it. The next candidate content then depend on the previous contents grade, and need to get a better grading. This type of algorithm that varies the result to find a better solution is called a genetic algorithm. [1, 2, 9]

The grading of generated content is often done by determining what qualities are important in the generated content. With that knowledge one can make a mathematical function that one can use for the grading. It’s also common to give different parts of the function different priority’s to help the grading be relevant. [9]

METHOD

Implementation
Since I have worked on my thesis following an agile methodology, the direction of my work has changed along the way. During development I implemented procedural generation in a few different ways, and I think it is important for my thesis to represent these different implementations and not just the final result. I have therefore divided the result section into three different implementations.

The first implementation focuses on generating a level that should be playable right away. The user simply has to press the play button and gets a new level to play. This would remove the burden of content creation from the developer completely, so it would be very valuable if it could be accomplished.

The second implementation instead focuses on generating a level that serves as a base for the level editor. This way, the user who likes to create content for the game gets a base to start building off. It helps the users create unique levels.

The third and last implementation focuses on helping the user in the level editor by providing tools to build the levels with. For example a brush to paint asteroid belts with or a tool to place asteroid spirals. This implementation is meant to help the user create more unique levels, like the second implementation.

Development Tools and Programming Language
The game that I worked on in this thesis had already been developed using Gideros Mobile, so that was what I would continue using. Gideros Mobile is a tool that allows cross-platform development of mobile games. Its main purpose is to help developers avoid having to write different code for the Android and iOS versions of a game. The code is written in Lua, and Gideros supplies extra game related base classes for physics and sprites. Lua is a scripting language that is considered very fast and it works on most platforms. This is because it runs as bytecode interpreted by a virtual machine. Because of this, Lua is often used as a scripting language for games. [5, 6]

Agile
While I have worked alone on my thesis, two other people worked on the game at the same time as me.

7 http://minecraft.gamepedia.com/Seed_%28level_generation%29
We needed to have some way to divide responsibilities, keep track of bugs and keep track of progress. We chose to use an agile\(^8\) work method in order to fulfill these goals.

We chose to use Trello\(^9\) as a form of SCRUM\(^10\) board, so we could easily keep track on the project no matter where we were.

We routinely discussed the project with our customer and changed direction when he felt it necessary. He also tested the game routinely to let us know what he thought worked well and what didn’t. One thing that we changed as a result of these discussions was that we removed repulsion nodes from the game, because we found out how easily they could be used to clear the path to the goal.

Since we worked in the same room and were only three people we didn’t have any kind of SCRUM meetings. Instead we continually discussed the project while we were working. [3]

---

\(^8\) http://en.wikipedia.org/wiki/Agile_software_development

\(^9\) https://trello.com/

\(^10\) http://en.wikipedia.org/wiki/Scrum_%28software_development%29

### RESULTS

**First Implementation**

On the first implementation my main objective was to be able to generate new levels on the fly. The user should simply click on the play button and the game should generate a functioning level. By using the distinctions described in the theory section, we can describe the desired level generation as online (because the content should be generated immediately) and necessary. It should preferably be dependent only on a random seed so it could be shared. When it comes to stochastic or deterministic generation it wouldn’t matter much since the generation would rely on a seed and the user wouldn’t input any parameters. Using the same seed would create the same result anyways, since the random function gives the same results every time the same seed is used.

When it comes to the testing, a constructive test would be best, since a generate-and-test approach is very time consuming. The generate-and-test approach also requires an easily searchable space, which Gravel doesn’t have. [4]

This implementation first randomized the position of the player and then placed it in the game world. After that the goals position was randomized and if it was at least a certain distance from the player it was placed. Otherwise a new position was
randomized until a suitable distance from the player has been generated. After that the asteroids were generated.

First the number of asteroids was determined by randomizing a number between a maximum and minimum value. A position was then randomized for each asteroid. If the position was free the asteroid was placed, if not it was skipped and the function moved on to the next asteroid. The number of gravity node power-ups that would be placed was randomized from 1 to 4 and also gave between 1 to 4 power-ups.

Second Implementation

The second implementation focused on generating a base level that could be used in the level editor. The user could choose one of three different ways for the generation to work. The first choice would generate the level as in the first implementation. The second way would cluster the asteroids around a few locations on the level. The third way would generate the asteroids in belts that snaked across the map.

The cluster generation functioned by randomly choosing a position and then save it. That position was used as the center of a circle which represented the cluster. Then asteroids were generated inside a square around that position. The squares sides were as long as the circles diameter. When each asteroid was generated, a check was made to see if it was inside the circle and that it didn’t overlap any other asteroids. If the asteroid fulfilled these criteria it was placed on the level and the function moved on to the next asteroid. If the asteroid failed to pass it was instead discarded.

The generation of the asteroid belts built on the cluster function. Basically it created a number of small clusters in a line that bent. It randomized a starting position, then randomized a new position around itself, and all the following positions after that were limited in which direction they were randomized. The belt could only bend a certain number of degrees in either way in order to prevent sharp turns, which looked strange and unnatural.

The goal and player was still placed by the generation function, but the user could still move all game elements around after they had been placed. The player could then use the base that was generated to build a better level. The user could move the goal to a position that was harder to reach, and the user could move the player to a situation that was trickier to get out of. The power-ups could be moved to positions which were more feasible to reach and the number of starting nodes could be adjusted better to the levels difficulty.

The user could also place a square shaped tool that could be used to generate more asteroids within that shape.

This kind of generation is a bit different from the

![Figur 2. Example of generation from the second implementation](image)
first implementation. It can afford to be slower since it is used in the level creation process. However, the level editor is meant to be used by the users, not the game designers. A game designer can deal with a slow level editor because that is his job; a user however only has so much patience. Therefore, this implementation was still coded to be relatively fast. This content being produced in this implementation cannot really be classified as necessary, since the user can just restart the generation a few times to get a desirable result. So this generation focused on optional content.

Since the levels were not totally generated, it would not be possible to share them with simply a seed. Since this meant that only using a seed was not a priority any longer, the generation was used parameters more. If I had spent more time on these implementation, the idea was that the user should have been able to feed parameters to the generation. The second implementation was therefore more parameter driven, as opposed to the more seed driven first implementation. The functions never generated the same things even when they got the same parameters, so the generation was stochastic.

The last distinction that changed in the second implementation was that since a user would generate these levels in the editor, the task of validating the result would then fall partly on the user. The generation still controlled the more simple things, like if objects were generated on top of each other. However, the user had to control if the level was possible to complete and how many gravity nodes were needed.

Third Implementation

In the third implementation the generation had been relegated to a more discrete role. The users would now create the level from scratch and then use procedural generation to add asteroids in a more directed way. This implementation made it possible to generate asteroids with a brush-like tool, so the user now had control over in which area the asteroids would be generated. This meant that the users could draw their own asteroid formations.

This worked a bit like the belt generation in the second implementation. Small clusters were simply generated along the line the user drew with his finger.

Another new addition was that the user could now generate formations while touching the screen. The user could chose to generate a spiral, and then press the area where he wanted the spiral to start from. The spiral then grew out of that area until the user lifted his finger. The user could also choose to generate asteroids in an asterisk shape instead.

Since this implementation is used for elements in the level editor, the generation still needed to be quick. The functions that were used never returned the same values so they can be classified as stochastic. They also use parameters and generated the seed internally, which means the generation was mainly parameter driven. The content generated was optional, since the user could delete it if the output wasn’t satisfactory. The generation was constructive, since validation still occurred in the function and by the user.

DISCUSSION

Results

First Implementation

The first implementation worked fine as a proof of concept but it also showed that there were several problems with trying to generate a level that the user could play immediately.

The first problem was that the generation created very boring levels. The first reason for this was that since the whole level was filled with asteroids randomly they became pretty evenly spaced. Therefore it was not too hard to finish the levels.

The second reason that the generated levels were boring was that they all looked almost the same, like the level in figure 1. While the asteroids were not in the same place, the levels almost always played the same.

Another problem with the first implementation was that the power-ups were not being placed very well. They were often either too difficult to pick up to be helpful, or they were too trivial to pick up and gave too much for such a small effort.

If I had not moved on to the second implementation though, I think that the problem with the power-up placing could have been partially solved, and I did do some preliminary work on it too. The solution I used in my preliminary work was to check the distance of the power ups to the player before placing them. If they were too close or too far away a new position would be generated. This improved the generation a bit, but it didn’t take the obstacles between the player and the power up into consideration. In order to create a better challenge the game would have needed to know if there were many objects in the way that would need to be bypassed.

Another, similar problem was how to determine how many nodes the player should start with. One could just give a number of nodes to the user depending on the distance to the goal, but that
might not be enough if there are a lot of obstacles in the way. And if one tries to take the obstacles between the goal and the player into account, one must also take into account the surrounding obstacles, since it is easy to just go around an obstacle if there is nothing there. Then it’s necessary to determine how to use the knowledge of the surrounding obstacles. Just because there are objects in the way doesn’t mean that it’s not relatively easy to create a path through.

This is part of the reason that the costumer and I felt that it was best to change direction. In order to properly validate every level and balance the difficulty the game would have to make many complex test runs. It would be much too slow to do on the user’s phones, so the levels would have to be generated by us and then pushed out to the users.

The reason that the testing would be so complex is that the game can’t easily be represented in a much simpler form than what is in the game. A level with five hundred asteroids would be quite difficult and taxing to test, since you would need to calculate how they interact with each other and bounce around after using a gravity node on them. Since the gravity nodes aren’t tied to the player, but instead can be placed anywhere on the map, there are a huge number of different viable moves that needs to be tested, too many for it to be feasible to test them all. The alternative would be to find a way for the computer to determine if a move is worth testing or not, but then my thesis would change focus towards AI instead of procedural generation. [1]

In order for a generate-and-test approach to work the game needs to have a simple game space. A dungeon would be a good example. Dungeons are easy to generate because you can easily represent the dungeons as expression trees. You can break it down to the fact that a dungeon consists of two things, rooms and halls. The halls connect the rooms together, and one can easily say that this room is connected to these halls, which in turn are connected to the following rooms or halls. The only thing that needs to be validated is if one can get from the entrance to the exit. Of course, there are many more things one wants to add to a dungeon, but those don’t require the same kind of validation. There is unfortunately no way to simplify Gravel in this way, which is why a different approach was needed. [2]

It could also have been possible to create a mathematical function to help evaluate a level by analyzing the level composition. The problem with that approach is that Gravel has too few boundaries. The level is basically limitless, which makes it complicated to determine the best composition of a level. It also requires more mathematical know-how then I possess. It could certainly be something worth looking at for someone with a better mathematical education. [9]

Another way to get around the problem of guaranteeing good content could have been to generate the levels without validating them, instead letting the users rate the levels and/or collecting statistics on how well players perform on a certain level. This information could then be used to filter out the levels which could not be completed. This would however provide enough work for another person to do a thesis on, and it’s doubtful that the players would want to act as testers without some kind of incentive. So we chose to try an implementation that needed less complex validation.

**Second Implementation**

In order to avoid the validation problem this implementation took the form of a generation tool for the level editor. So instead of the computer needing to check if the level was playable and challenging, the users could take over that responsibility.

This implementation improved much on the lack of variety that the first implementation suffered from by using the clusters and belt generation. These generation methods created more natural results, and with some work they could have look even better. The implementation was not without problems however.

First of all; just because the generated levels looked more natural, did not mean that they were interesting to play. The generation with belts and clusters was definitely more interesting than the completely random generation done in the first implementation, but it still didn’t create very complex levels. It all came down to just swinging the player ship around asteroid clusters. To create an interesting challenge one had to generate a base several times. And even then it still felt like it lacked the challenge a handcrafted puzzle could provide.

The way I would have liked to tackle this problem would have been to give the user more control over the generation, and several tools to help with small scale generation. Also, more complex problems could be constructed if the user hade more asteroid belts and clusters generated. This way, the user would have to complete a more intricate challenge, and it would allow for several solutions to a level.

This option was not possible though due to the fact that we were making a mobile game. When we created too many sprites on a level, around a
thousand, the game would run horribly, especially on older phones. It didn’t really seem to matter how large the textures were, using lower resolution images would only increase performance by a few frames per second.

The fact that we were making a mobile game also made it difficult to give the user a lot of options for how the generation should work. To make generation functions that could produce a wide range of different results, the user could have been allowed to send many parameters to the generation, for example the minimum and maximum number of clusters to be generated. It’s hard to make an interface for collecting all these kind of parameters on a phone though. It would feel very fiddly and most users would probably not even bother.

The described implementation would probably work better on a PC as number input is much easier and it’s possible to have a complex menu system. The costumer however wanted another approach for Gravel.

**Third Implementation**

This implementation of procedural generation is probably not what most people think about when they hear that term, but it is still procedural. The user is just given much more direct control over the generation. When the user uses the brush, asteroids of a random size and angles are created in random positions around the user’s finger. The size and angle of the asteroids are generated in the shape generation as well.

This implementation is undoubtedly a much more trivial solution then the previous implementations, but it fits the medium the best. Using the brush to paint asteroid belt work much better on a mobile then fiddling around in menus, and creating the shapes by just holding down your finger is also an interaction that works well on a mobile.

In figure 3 we can see the function that is used by both the cluster generation in the second implementation and the brush in this implementation. The interesting part is that since the result can’t be validated afterwards, we have to do a lot of validation during generation, showing that this algorithm is constructive. Without the code making sure that objects are not overlapping this function could be cut down to half the lines.

This does create some conflict between two of our desired qualities in this algorithm though. The code needs to make sure that objects aren’t generated on top of each other which is done by looping through all asteroids on the level. This can be very slow though when you have many asteroids on a level, which impacts the requirement to be fast.

There might be a better way to store the asteroids. If I had stored them in a way that would make it possible to search for them based on their location, it could have been quicker to find overlapping

```javascript
function Generation:generateCluster(numberOfAsteroids, radius, originX, originY)
for _ = 1, numberOfAsteroids, 1 do
    local x, y
    repeat
        x = math.random(radius - radius, originX + radius)
        y = math.random(originY - radius, originY + radius)
        until math.sqrt((x-originX)² + (y-originY)²) <= radius
    local size = 0
    while size < 0.3 do
        size = math.random() * getSizeMultiplier()
    end
    angle = math.random(0, 360)
    local conflict = false
    if next(self.playerCords) == nil or math.sqrt((self.playerCords[1]-x)² + (self.playerCords[2]-y)²) >= 100 then
        if next(self.goalCords) == nil or math.sqrt((self.goalCords[1]-x)² + (self.goalCords[2]-y)²) >= 160 then
            conflict = false
        else
            conflict = true
        end
    else
        conflict = true
    end
    if not conflict then
        local image = Bitmap.new(Texture.new("Sprites/Asteroid.png", true))
        local newRadius = (image:getWidth()/2 * size) - image:getWidth()/10
    for index, cord in next, self.cords, nil do
        if math.sqrt((cord[2]-x)² + (cord[3]-y)²) <= cord[3] + newRadius then
            conflict = true
        end
    end
    if not conflict then
        self.world:addChild(Asteroid.new(self.world, x, y, "Sprites/Asteroid.png", size, angle))
        table.insert(self.cords, {x, y, newRadius})
    end
end
end

Figure 3. Code example of the cluster generation function
asteroids. But as long as not too many asteroids are placed on the level, this is not a problem.

**Method**

*Agile*

Using an agile work method has worked well for this project because it wasn’t very clear from the beginning in what form procedural generation should be implemented. The agile work method allowed me to discuss the issues with the customer as they arose, and it allowed me to change direction when it was necessary.

Using a SCRUM board worked fine for keeping track of bugs and responsibilities in the development of Gravel. Without it we would have had trouble remembering rare bugs, and might have left them in the game. When we encountered bugs that we had yet to fix, it was helpful to be able to collect any new information on the bug in the same place. This made it much easier to get up to speed on certain bugs when it was time to fix them.

The fact that all three thesis workers sat in the same room helped immensely. This allowed us to quickly discover bugs that one worker had caused by working on his part of the game. An example would be that someone changed how a function in the asteroid class worked. When we were sitting next to each other it was easy to find out what had been changed exactly. It also allowed us to brainstorm together.

There have been some downsides to working agile too though. While it has been necessary to be flexible in the implementation of procedural generation, it is also a bit frustrating to constantly have to change direction. The final result might have been more complex and interesting if I had done more research at the start of the project and then had a discussion with the customer of what most likely would have worked and what would not work. I could probably have skipped the first implementation completely if I had done this.

**Implementation**

The biggest issue with the implementation was that the code became quite unstructured and messy, which is partly my fault and partly the fault of Gideros and Lua.

Lua has quite a few issues that bothered me and I believe leads to poorly structured code. The first issue is that Lua doesn’t default declared variables as local, instead they choose to default declared variables a global. So unless you want to use global variables for everything, which would create very confusing code, you need to declare the variables as local. The first problem with this is that it’s really easy to forget, since almost no other language works this way. The other problem is that it looks messy when you use the same keyword all over the place for variable declaration. To make things worse, variable declaration is identical to assignment, so it is easy to create a new variable instead of assigning a value to an existing variable.

This issue is made worse by another issue, which is that referring to variable that doesn’t exist doesn’t return an error, it just returns nil. That means that misspelling a variable name or using it outside its scope will not throw an error until you try to run the code in question. And then it will only throw an error if the code in question doesn’t allow a nil value, otherwise you will just get some strange results.

```
local counter = 0
while counter < 2 do
  counter = counter + 1
end
```

The above example would cause an infinite loop, but it would not cause an error when trying to run it. The interpreter would just assume that you wanted to create a new variable named “counter”.

The number of times that I have had code not work or work weirdly because this are numerous and it slows work down.

Gideros has a pretty big downside that needs to be addressed. Lua does not provide an easy way to use classes, but it can be extended to allow them. Gideros has done this, but the way classes work leaves a lot to be desired. There is no straightforward way to make class variables and functions private, instead everything is public. There are workarounds for this, like making a function in a class a local function, but it would have been much better if functionality for this was actually built in. This also led to some badly structured code.

In general, a language that gives the user more control and allows for a stricter structure, like C++, would probably have worked better for the more complex code I wrote for the first and second implementation. It would have made it much easier to create a more readable and less convoluted solution.

**CONCLUSIONS**

The purpose of this thesis was to explore if there is any way to implement procedural generation in the game Gravel, which I have done. I have implemented three different types of procedural
generation: generating a whole level, generating a level to use as a base in the editor and generating elements on the fly in the editor.

Of these, the third implementation - generating elements in the editor, worked the best. The reason for this was that the interactions could be made easy on a mobile phone. If Gravel had been a pc game, a combination of the second and third implementation would probably have been best. This is because the main drawback of the second type of generation was that it required a lot of input from the user in order to produce interesting results. This would not work well with a touch based user interface, but would be made easy with a mouse and keyboard.

The first implementation is not really viable for this kind of game, because it requires a generate-and-test approach. This approach is difficult with Gravel because it’s there are so many moves that could be made that it would be hard to test them all. Also, it is not enough to check if there is a path between the player and the goal, since obstructions can be removed with in-game elements. Since it is a puzzle game it’s important that levels are difficult too, which is also difficult to determine.

REFERENCES


2 van der Linden, Roland, et al. "Procedural Generation of Dungeons", IEEE Transactions On Computational Intelligence And AI In Games, VOL. 6, NO. 1, MARCH 2014


5 Lua (programming language), http://en.wikipedia.org/wiki/Lua_(programming_language)

6 Lua, About http://www.lua.org/about.html


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Björn Jansson