WorldObjects Code Review:

When we first started thinking our “must haves” for this project, one of the first ideas was to have a main class WorldObject. This class basically represents all of the objects in the game, like the obstacles, the player, but also the floor. Every WorldObject object has a texture and a position in the world. In the current version of the code, we have three different WorldObjects: Player, FloorTile and Obstacles.

During the implementation of the Player, we ran into the problem that we wanted to change its texture, as well as adding an animation to the player. We did this using a different class, AnimatedWorldObject, and created animation and texture changes in that class. Due to this change, instead of making Player a subclass of WorldObject, we had to make AnimatedWorldObject a subclass of WorldObject and then make the AnimatedWorldObject the superclass for the Player. This change didn’t create many issues, besides changing the constructor for the Player.

Another big change in our code has taken place this week. We changed the Player to work with the main game library that we are using in this project, libGDX. This modification made us rewrite some things, especially in the constructors of the WorldObjects.

Physics implementation:

Our physics have been all over the place from the very beginning. What started off as a quick and dirty implementation which failed at collision detection when speeds got too high, and had rather counterintuitive jumping behavior, slowly turned into a mess that made death impossible and bugged out at random moments. Collision detection functioned, but was implemented in an inflexible way that made it hard for us to detect whether an object that the player collided with should cause death.

The time had arrived for a large overhaul, and that’s been one of the points of focus for this sprint. First of all, we replaced our custom collision detection logic with a Box2D-based system. Not only are the physics a lot more realistic and does collision detection work right, it is also possible to detect which objects the player is colliding with, making it possible to check if the player should die using a very simple heuristic. On top of that, it has become a lot easier to tweak the behavior of the physics by just changing a few parameters.

It would be going a bit far to say that the physics are fully where they need to be (for one thing, the player tends to tumble quite a bit at the moment, and bounding boxes are a bit too large at the moment), but it is already a great improvement from what it used to be before.

Generated Levels:

The main functioning of the generated level as discussed in the requirements is to be:

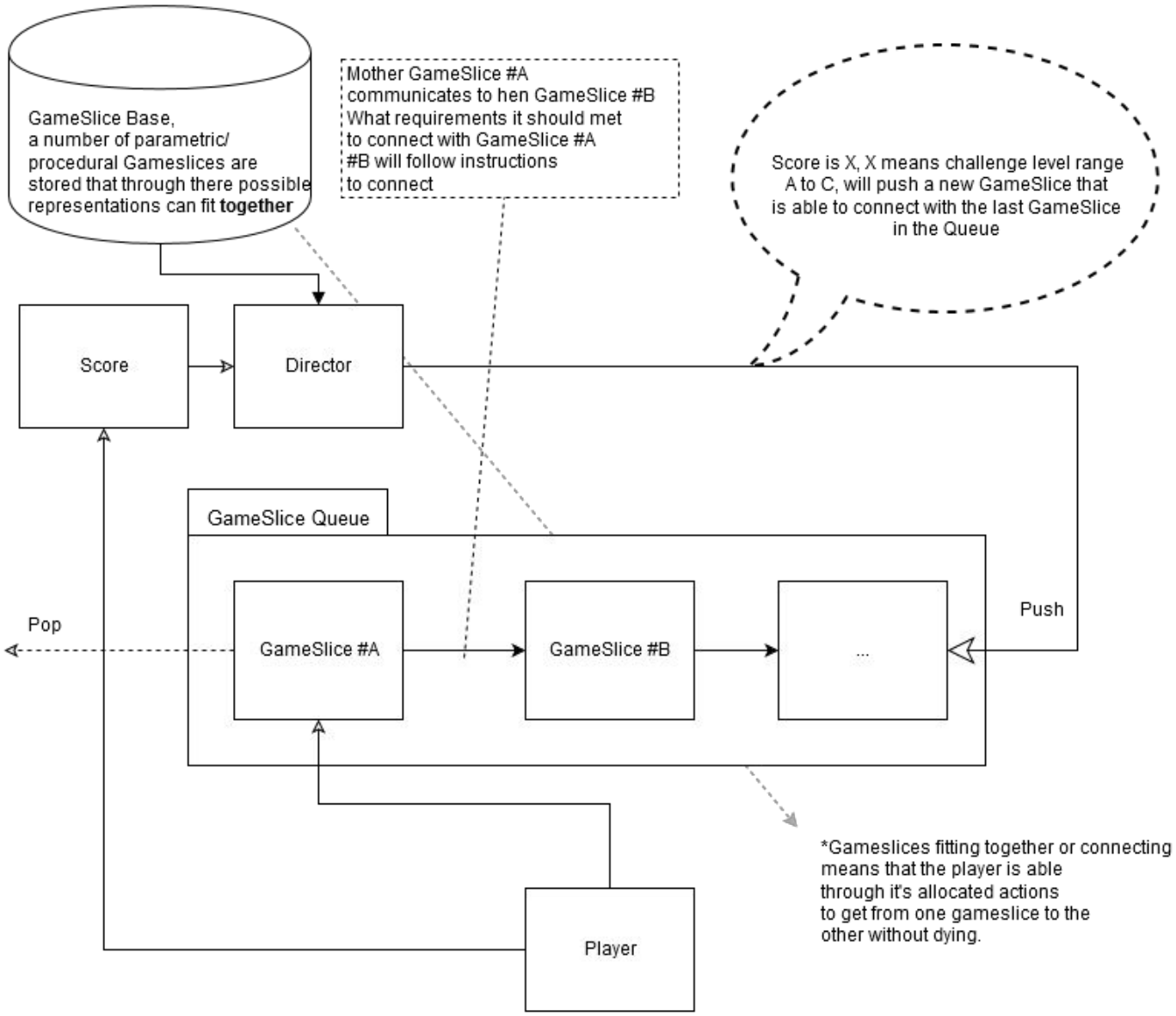
- infinite
- getting harder as the game progresses
- procedural

In this short text it is discussed if the current implementation is optimal to extend to these requirements.

In short a level consists of a queue of **Gameslice**s that the player must traverse. These **Gameslice**s are put in the queue by the **Director**. In the director logic can be put what slices would be interesting to put in next.

The things that I find good currently is that there is a director class that is able to randomely do things.
The things that I find can be better is that the director class does too much. The director should put gameslices into place, but the gameslices should decide for themselves how they configure themselves. The gameslices are actors that have to work together to create a coherent whole. So then there should be a few different gameslices that configure themselves differently have different elements or configure them in different ways. One gameslice can be about a climactic experience, a few little hops following a big jump, an other gameslice can be about just some random blocks etc. To allow for these different things Gameslice should have children that describe each one of such interesting elements. And very importantly they decide themselves their configuration, and talk with the gameslice that is closer to the player what they are able to be for the player to be able to progress.

The way I would like it to be: (see image on next page)


GameSlice Base,
a number of parametric/
procedural Gameslices are
stored that through there possible
representations can fit **together**
Mother GameSlice #A
communicates to hen GameSlice #B
What requirements it should met
to connect with GameSlice #A
#B will follow instructions
to connect
Score is X, X means challenge level range
A to C, will push a new GameSlice that
is able to connect with the last GameSlice
in the Queue
Score
Director
GameSlice Queue
Pop
GameSlice #A
GameSlice #B
...
Push
*Gameslices fitting together or connecting
means that the player is able
through it's allocated actions
to get from one gameslice to the
other without dying.
Player