Dorf Fortress

Brief Description

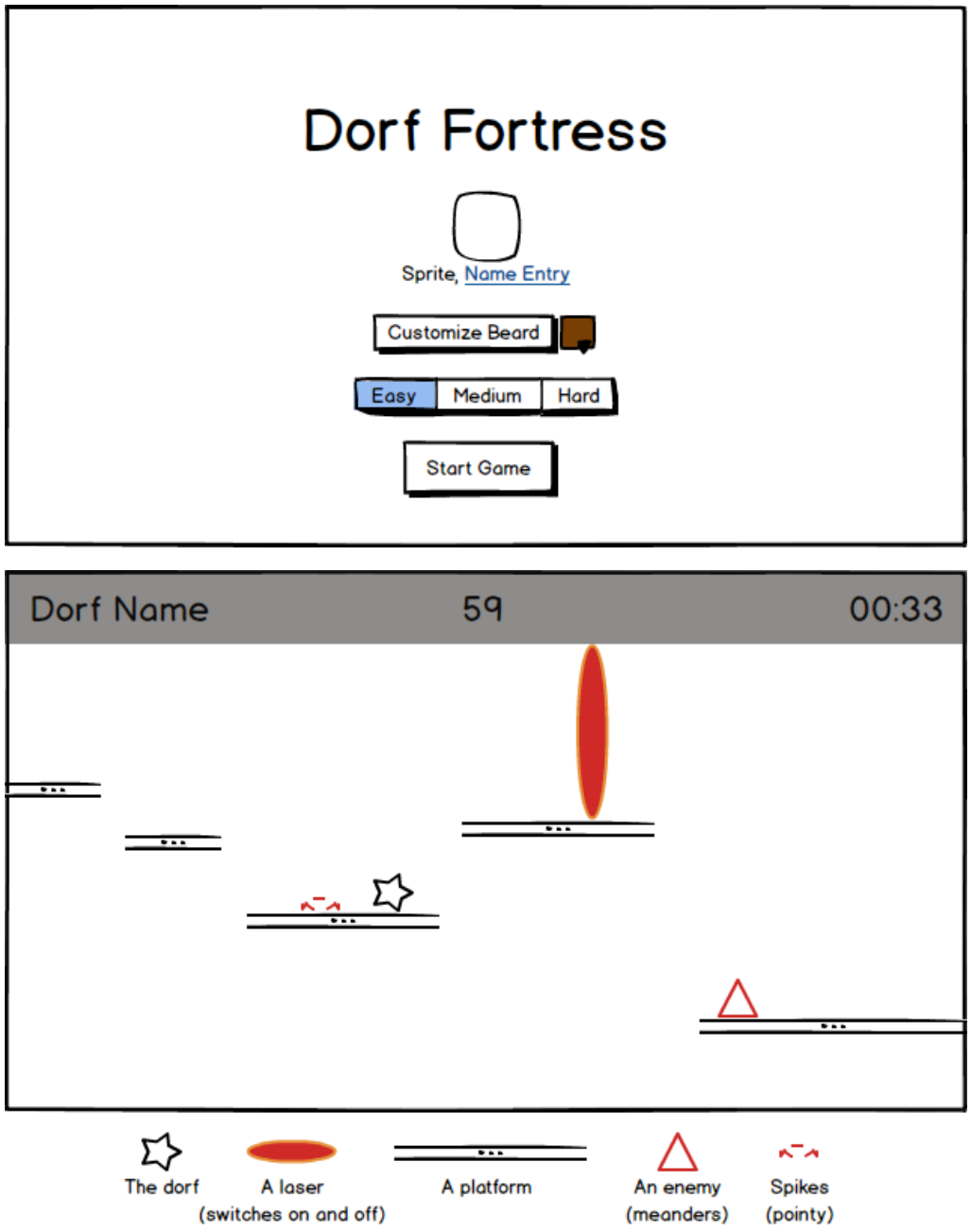
Our program will, at its core, be a fairly rudimentary platformer game; the player will control a dwarf (rather, a dorf, as we enjoy strange misspellings) as he explores an abandoned castle. Our larger objective is to procedurally generate levels for the game, while making sure that the levels are possible to complete. This will be a significant challenge for us, so we're expecting that the platformer itself will be relatively simple, though hopefully still appealing and polished. Our stretch goals mostly involve adding features to the platformer itself - different types of obstacles and enemies, moving platforms, character customization, etc - which will then be procedurally generated as well.

Feature List

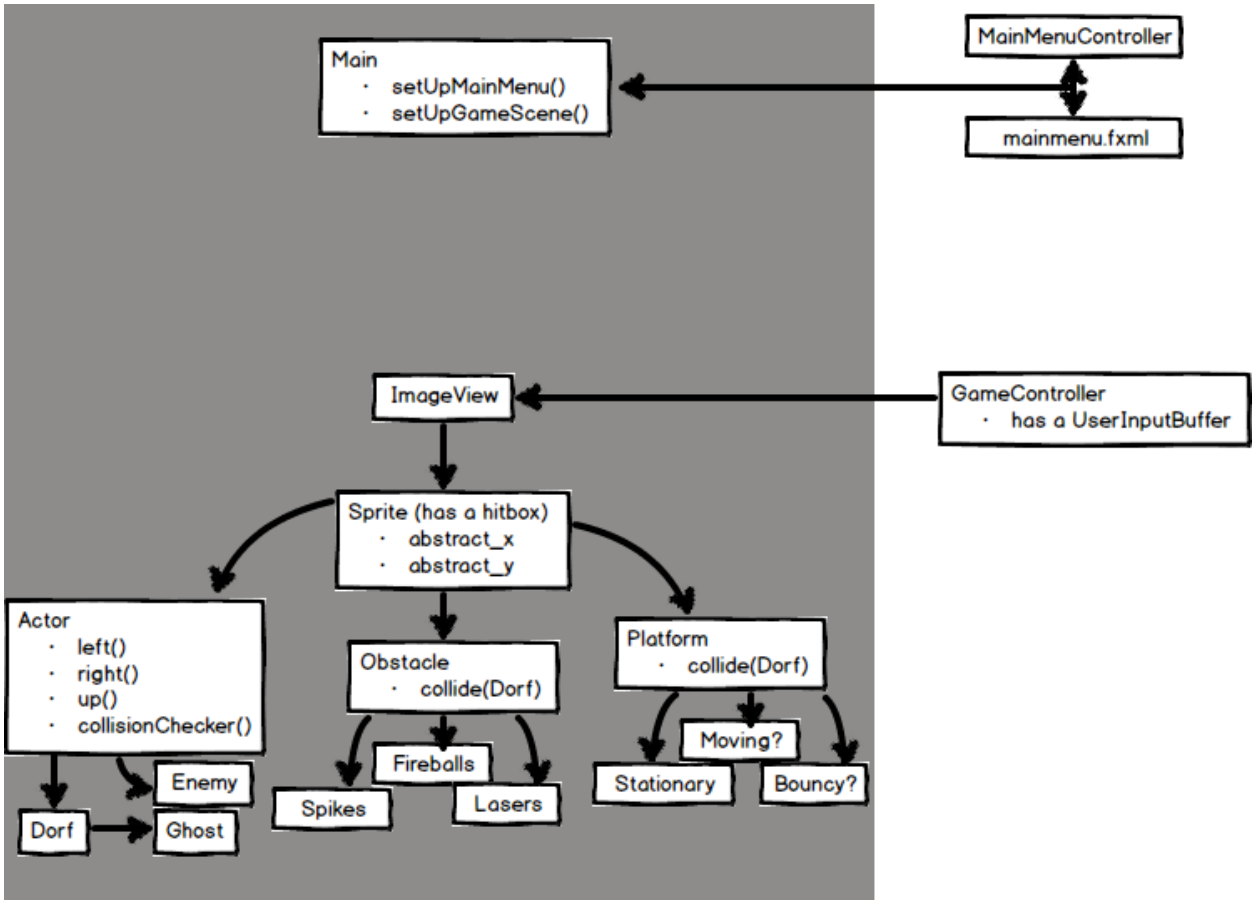
- Basic platformer functionality - the character collides successfully with platforms, can move around a hard-coded sandbox level and jump with WASD/Space/Arrow keys.
- Static obstacles that kill the character on collision.
- Functionality for moving obstacles.
- Side-scrolling functionality; a level larger than the application's window that pans to follow the character.
- Procedurally generated platforms that provide a possible route from the start of the level to its end.
- Procedurally generated obstacles that provide difficulty while not actually intersecting with said possible route.
- Multiple obstacle types
- Moving platforms
- Nice-looking sprites and graphics
- Enemies (these are hard to account for given the method we're using to generate levels, since their motion is a little less predictable and they can be killed, so they're a little trickier to implement than the obstacles.)
- Animated sprites

- Character customization (i.e. beard color, hair/helmet style; we found a fairly large sheet of different dwarf sprites, so we have the resources to do this)
- Keeping track of where the character dies in a level, and spawning enemies there on future playthroughs.
- 3D graphics with Oculus Rift support

Mockup



Software Architecture Diagram



Component Signatures

MainMenuController

- Click methods for each button

GameController

Variables:

- int screenXShift - is used in displaying object for side scrolling
- Platform platforms
- Obstacles obstacles
- Ghost ghost
- Dorf playerSprite

- UserInputBuffer inputSource (gettable)
- Parent root

Methods:

- (observable methods are used to pass screenXShift back and forth to other objects for displaying)
- **Initialize**()
    - The animation loop is here general order of things
        - Run()
        - inputSource.reset()

UserInputBuffer

This object is what stores user input for a given frame time, it will ignore duplicate input during a given frame and it will clear during the animation loop of each frame, this helps modularize the access to key inputs as well as handle the KeyEventHandler logic

Variables:

- Dictoinary - stores key user input pairs

Methods:

- **addKey**(keyinputobject) - adds input to the dictionary and doesn't doesn't produce a duplicate
- Boolean **wasKeyPressed**(keyinputobject) - returns a boolean representing whether the key was pressed during the current frame, used for key controllable objects.
- void **reset**() - clears the buffer for the end of the animation time

Sprite extends ImageView

Variables:

- Hitbox hitbox (gettable)
- Has *abstract* xy coords (relative to start, because it's a scrolling level). Subtract the **Dorf**'s coords to get an xy on the *screen*.
- screenOffsetX - is used to draw objects in their correct place on the screen
- int abstractX (gettable and settable)
- int abstractY (gettable and settable) - the abstractX and abstractY are the "true" coordinates of the object in the simulation. For display purposes the setter will update the X and Y coordinates for the JFX object so that it displays properly on the screen. Will also update the x and y coordinates for the hitbox when it gets updated.

Methods:

- **Update**() (part of the observer interface) - used by game controller to change screenOffsetX which then uses the abstractX and abstractY values to change the “real” x and y values for the object.
- void **collide**(Sprite sprite) - deals with the various collision behaviors that the class enacts upon something else that hits (e.g. platforms modify the player to have the

Hitbox extends **Shape**:
class that that consists of a shape that isn’t attached to any particular root. The the size and shape of the object can be handled by subclasses that use the interface

Boolean **Intersects**() - determines if the two shapes overlap at all.

Actor extends **Sprite**

- Subclasses: **Dorf**, **Ghost**. Superclass: **Sprite**.
- void **step()** - overridden by each implementation
- Sprite **checkCollision()** - Checks for **Platform** collision, **Obstacle** collision, and returns the object collided with.
- void **die()**

Dorf extends **Actor**
Variables:

- GameController stage - Stores reference to the stage controller so it can access user input loop
- Boolean onPlatform - used for jump phusics
- (various other booleans and information used to handle movement, variables for double jumps and whatnot)

Methods:

- void **step**() - handles moving the object based on its velocity and adding any gravity whatnot and moves the object. This will move the GameController’s screenXShift so the camera follows the dorf
- void **applyUserInput**() - accesses the user input buffer in the class and makes the necessary changes to the the movement object

Obstacle extends **Sprite**
An interface for a tree of interchangeable subclasses which defines the behavior of that object.
Variables:

- killBox - an object space on screen that will result in collision

Methods:

- void **step()** - moves/expands/changes character of object based on system time (this can have a very wide variety of behaviors for different classes of objects, ex spinning blades and whatnot)
- boolean **doesIntersect**(sprite s) - takes a given sprite at a given time that has a hitbox and tests it against the current hitbox for this object (e.g. is the given sprite's hitbox intersecting with the path of the laser?)
- void **Place**(hitbox h) - takes a given hitbox location

Platform extends **Sprite**

- **collide**(Sprite sprite) - for platforms it handles making the other sprite "attach" to the platform properly
- we could possibly implement moving platforms, but it's not essential

Procedural Generation Stuff

ObstaclePlacer

Class takes a given level and ghost object and iterates tries placing objects in the level, then iterates through the object hitboxes and ghost hitboxes to determine if a given obstacle will end up impacting the ghost, if so, it will delete the offending hitbox and try again

Variables:

- GameController Stage
- Ghost

Methods:

- List<Obstacle> **getObstacles**(number n) - calls placeObstacle(n) then cull obstacle for the resulting list then it stores the results. If the resulting list contains less than 10 obstacles, it will call placeObstacles(n - listsize) and then cull them again recursivly until it has enough valid obstacles to place.
- Obstacle **randomObstacleFactoryMethod**() - contains a list of every obstacle object that exists, will select an obstacle randomly and constructs it.
- List<Obstacle> **placeObstacles**(number n) - chooses random frames between the first frame and the last frame of the simulation according to the Ghost and iterates through the ghost frame hitbox and places n obstacles such that their path comes close to the ghost
- List<Obstacle> **cullObstacles**(List<Obstacle> obstacles) - iterates through the ghost frames and the hitbox returns for each obstacle it is given and determines if it's accessible

GhostFrame
Class that stores and mimics small chunks of user input into the game to run through the ghost, (e.g. one second's worth of frames holding down the right arrow key)

Ghost inherits from **Dorf**
Generally this class is responsible for ensuring that we build a solvable level as well as proving to the player afterwards that the level was possible by displaying the computer sprite solving the level
Variables:

- GameController Stage
- list<GhostFrame> a list of premade objects that correspond to randomly generated fake user input that is used to make the ghost's movements in simulation
- int **endFrame** - an integer corresponding to the final frame of the ghosts solution path (used to determine random placement of objects)

Methods:

- Iterator **getHitboxes()** - For a given stage returns a collidable rectangle object for the frames to level completion in order so that collision detection can be done on them while procedurally placing obstacles
- void **step()** - very similar to the step method for the super class except that before it carries out the superclass step() method it uses its list of GhostFrame user inputs to "add" its inputs to the InputBuffer

---

Tests will be made for...

- ObstaclePlacer
    - Are obstacles placed within an acceptable range? Generate a bunch and test.
    - Do they correspond to appropriate movements? Generate a bunch and test.
    - Are obstacles always placed appropriately? (e.g. Generate spikes and ensure they land on a platform.)
    -
- Ghost
- Obstacles
- Platforms
- GameController
- GhostFrame
- MainMenuController
