Bubble Beam - Assignment 5

Bavdaz, Luka 4228561
Clark, Liam 4303423
Gmelig Meyling, Jan-Willem 4305167
Hoek, Leon 4021606
Smulders, Sam 4225007

October 23, 2014

1 20-Time, revolutions

1.1 Multiple game modes

In the previous sprint we introduced serveral game modes (from requirements `M-191` to `M-194`). A Game Mode basically defines what kind of bubbles a player can receive in his cannon. We've implemented several *Power-up Bubbles* (bubbles with a special effect). These Bubbles are constructed through a `BubbleFactory`, and the Game Mode was basically implemented by providing the `GameController` with another `BubbleFactory`.

Ofcourse it's a bit ambigious to let the `BubbleFactory` be the object that decides the `GameMode`. It also was a bit limited: we could provide new bubbles, but a more advanced game mode - Timed Game Mode (`M-193`) - actually failed because we had no possibility to hook on to the required methods - translating the bubbles - and events - time.

Speaking of events, over time, the game controller logic became a bit cluttered, after adding hooks and observers/listeners in various ways. Thus, in this sprint, we refactored the event handling system as a starting point for the more advanced game modes and multiplayer improvements.

1.1.1 Event handling

We already use event handling a lot: the `CannonController` triggers a `CannonShootEvent` (which is itself most likely triggered by an `MouseEvent`). The `GameController` listens for this `CannonShootEvent` and then starts doing its responsibility: allow the `MovingBubble` to move and check if it collides with other bubbles on its way, and if so, handle this collision in terms of snapping to the `BubbleMesh`, or popping with other bubbles.

All these actions are in fact events as well, and provide perfect hooks for game mode implementations and synchronization in the multiplayer. However, in the current version, this eventing system was just not complete enough to make this true. Luckily, the changes don't require a lot of new classes to be introduced, but rather requires to move around a few methods between classes and update their callers.

BubbleMesh

The `BubbleMesh` is a data structure for the `Bubbles`, and this structure needs to be maintained as bubbles gets snapped in to the mesh, popped, or inserted at the top. These events can be useful, as points needs to be rewarded when bubbles pop. Also, when rows get inserted to the mesh, we want to send this to a potential multiplayer client.

`BubbleMesh`

Responsibilities	**Collaborations**
Datastructure containing the bubbles	`Bubble` objects
Logic to insert a new row of bubbles	`BubbleMeshListener`
Logic to snap a bubble into the mesh	
Logic to see if a snap caused any pops	
Notify `BubbleMeshListeners` of above events	

Figure 1: CRC-Card for the `BubbleMesh`

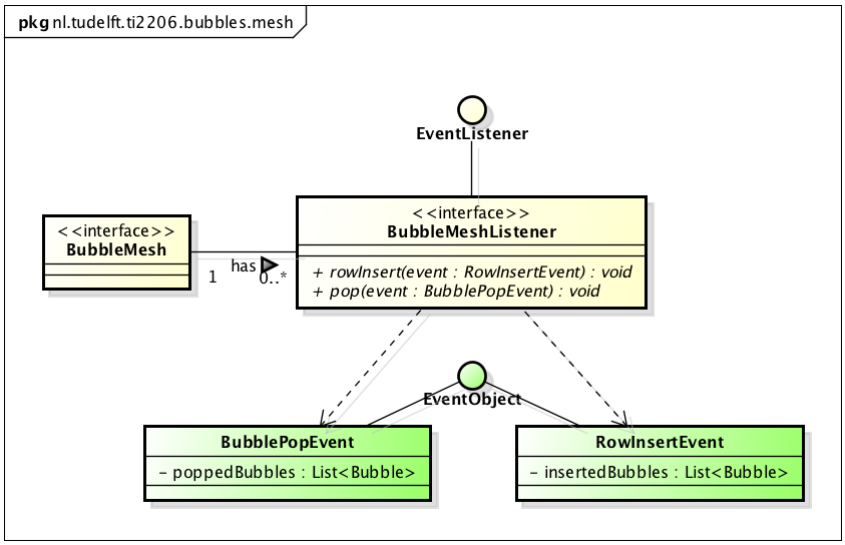

Figure 2: UML Diagram for the `BubbleMeshListener`

CannonController

The `CannonController` is responsible for the cannon specific logic. It triggers an event when the cannon shoots, which for example is necessairy for the `GameController` to start translating the shot bubble.

`CannonController`

Responsibilities	**Collaborations**
Updating the `CannonModel` when the cannon rotates	`Cannon` instance
Preventing new shoot while shooting	`CannonModel`
Propagating `ShootEvent` to the `CannonListeners`	`CannonListener`

Figure 3: CRC-Card for the `CannonController`

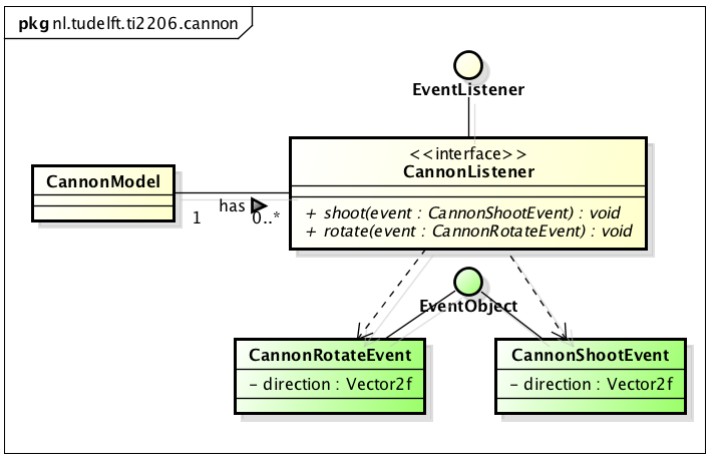

Figure 4: UML Class diagram for the `CannonListener`

GameController

The `GameController` is responsible for the generic game logic. See also the following CRC-card:

`GameController`

Responsibilities	**Collaborations**
Check collisions shot bubble	`BubbleMesh`
Update cannon ammunition	`CannonController`
Keep track of remaining colours	`GameListener`
Game Over handling	
Notify `GameListeners` of above events	
Propagate `ShootEvents` and `BubbleMeshEvents`	

Figure 5: CRC-Card for the `GameController`

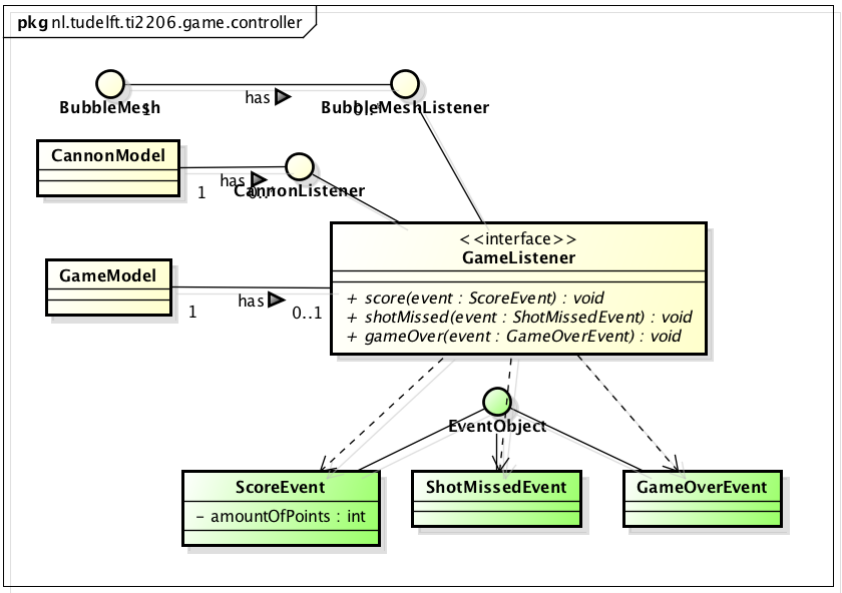

Figure 6: UML Diagram for the `GameListener`

1.1.2 Game Mode

From the requirements M-191 to M-194 we expect a Game Mode to have the following abilities: (1) it should be able to provide a certain BubbleFactory to the GameController, so that it can create the correct ammunation for the game mode; (2) it should be able to listen for GameEvents, for example to award points or insert new bubbles after a few misses; and (3) it should be able to listen on GameTicks to perform changes over time, such as pushing bubbles slowly to the bottom in the timed mode. Furthermore, we need to have access to the GameController to invoke these actions, and we also need to add some calls to the GameModel in the GameController.

GameMode

Responsibilities	Collaborations
Provide a BubbleFactory	BubbleFactory
Listen for GameEvents	GameController
Interact with BubbleMesh	BubbleMesh
Interact with GameControler	GameTick
Ability to hook onto GameTicks	

Figure 7: CRC-Card for the GameMode

Since we want a GameMode to hook onto GameEvents, we decided it should be a GameListener. Because we also want to hook on GameTicks, we decided a GameMode should also be Tickable. For the BubbleFactory and name of the GameMode, we defined getters in the GameMode interface.

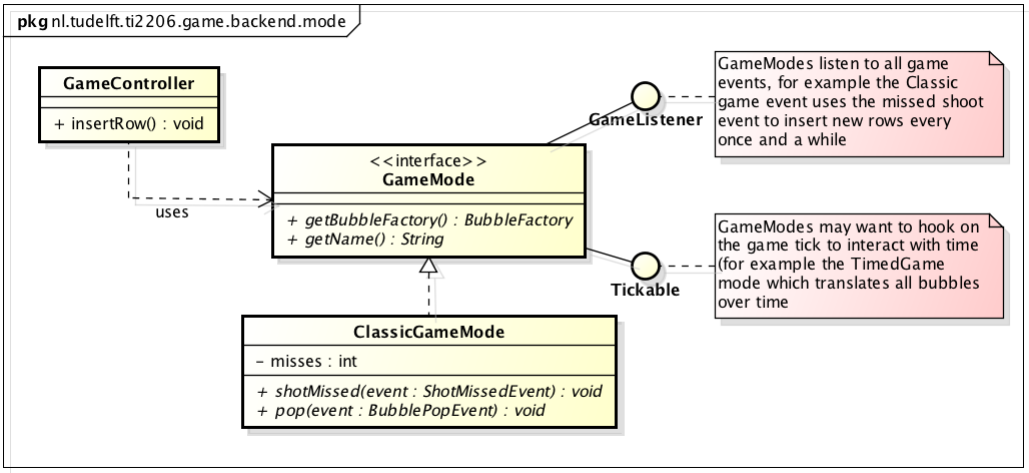

Figure 8: UML class diagram for the GameMode

Interactions

The ClassicGameMode provides bubbles through the DefaultBubbleFactory (which creates only ColouredBubbles and no Power-up bubbles), this is provided through the getBubbleFactory method. When bubbles pop, the player is awarded some points. This is achieved by overriding the pop method from the GameListener. When a shot bubble snaps into the BubbleMesh without popping, it's concidered a miss. After a few misses, a new row is inserted. The same as with the pop, this is done by implementing the shotMissed method. See also the sequence diagram 9 for these interactions between the GameMode and the GameController.

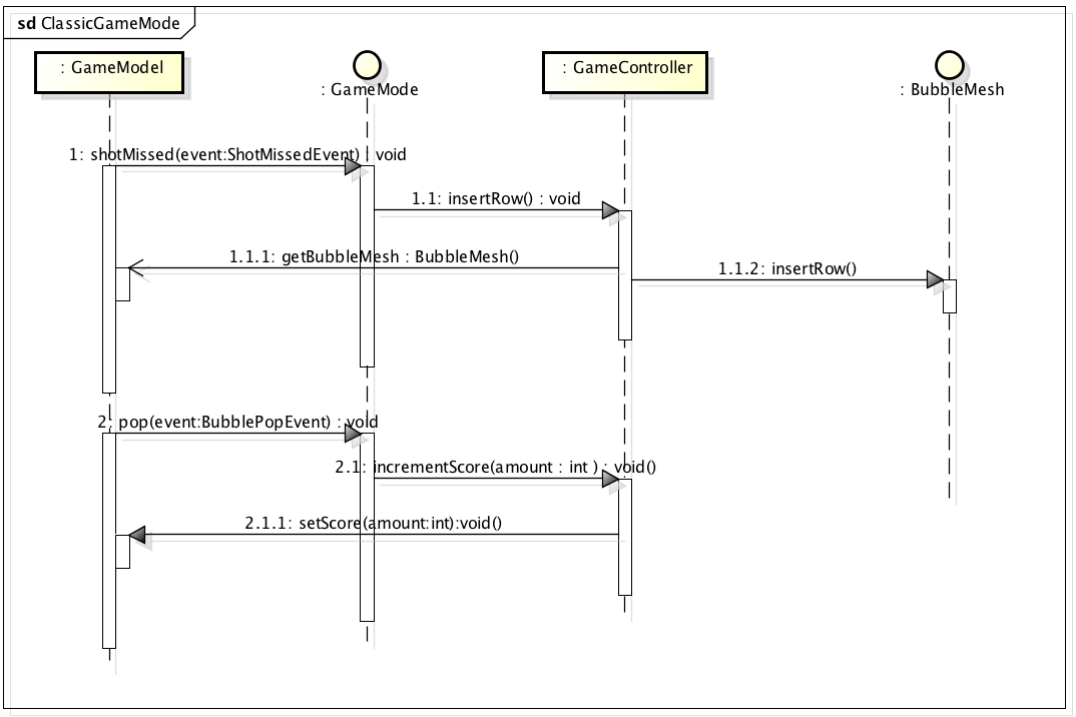

Figure 9: UML sequence diagram for the `GameMode`

1.1.3 BubbleMesh improvements

In the previous iteration a `Bubble` knew it's position and was able to paint itself on a `Graphics` object. In the `paintComponent` function of the `GamePanel` we iterate over all bubbles in the `BubbleMesh`, and invoke the `render` method. For the `TimedGameMode`, this was not enough. In the `TimedGameMode` we want all bubbles to slowly fall down at a certain speed. When they reach the bottom, the game is over, or when the `BubbleMesh` is empty, you have defeated the game mode. We needed to be able to translate the entire `BubbleMesh`.

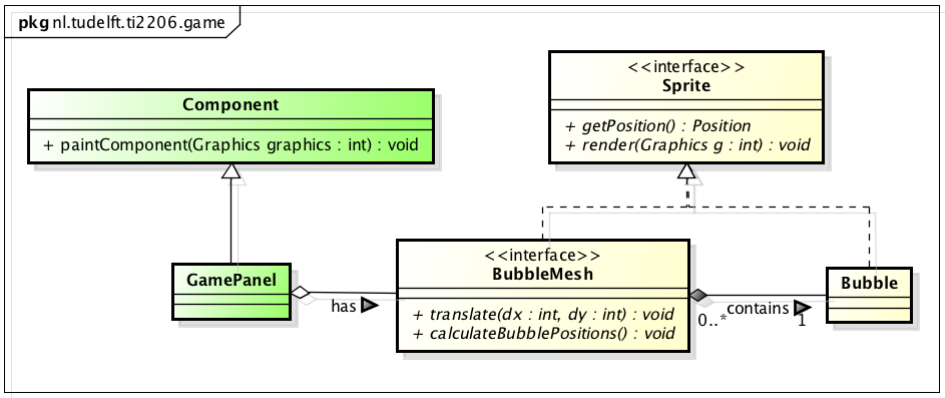

Figure 10: UML class diagram for the `BubbleMesh`

We figured out that it would be more clear to let the `BubbleMesh` be a `Sprite` as well, and give it the ability to draw itself and it's bubbles. Then the `GamePanel` calls the render method of the `BubbleMesh` instead of the induvidual bubbles. Also we gave the `BubbleMesh` a position which can be translated - also moving the bubbles in the `BubbleMesh`.

With these adjustments to the `BubbleMesh` and the event listener changes described in section 1.1.1 and 1.1.2, we now have all the ingredients to make the `TimedGameMode` work: in the `GameMode` we can now hook onto the `GameTick` and then slightly translate the `BubbleMesh`.

1.1.4 Game modes for multiplayer

In the previous version we basically only sended the `CannonEvents` and some `BubbleMesh` syncs, and let the client then guess what other events might have been triggered. Also, we just hard coded to always pick the `PowerUpBubbleFactory` (so what now would be the `PowerUpGameMode`). This did not give us the ability to play other `GameModes`, or use any of the Game Mode logic introduced in section 1.1.2.

Therefore we decided to rework the multiplayer. First, when we create a room (player 1 clicks "start multiplayer"), we want to be able to select one of the `GameModes`. Then we want to create two `GameModels` with this `GameMode` and an initial `BubbleMesh` and ammunition `Bubbles`. When a client connects (player 2 clicks "find multiplayer"), we need to transmit and process this initial data.

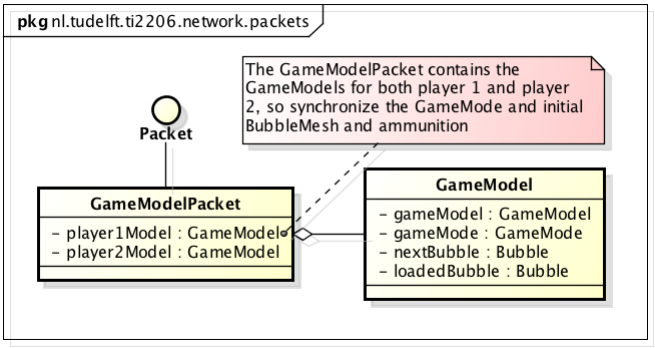

Figure 11: UML class diagram for the `GameModelPacket`

After transmitting the `GameModelPacket` both players can start playing. Now we need to transmit all actions between the two clients. In the previous version we used some dedicated packets for this, but the implementation was incomplete. Now we have an advanced event handling system (section 1.1.1), and all we have to do is listen for a `GameEvent` being triggered in the active game panels, wrap it in a `EventPacket`, transmit it to the client (figure 12).

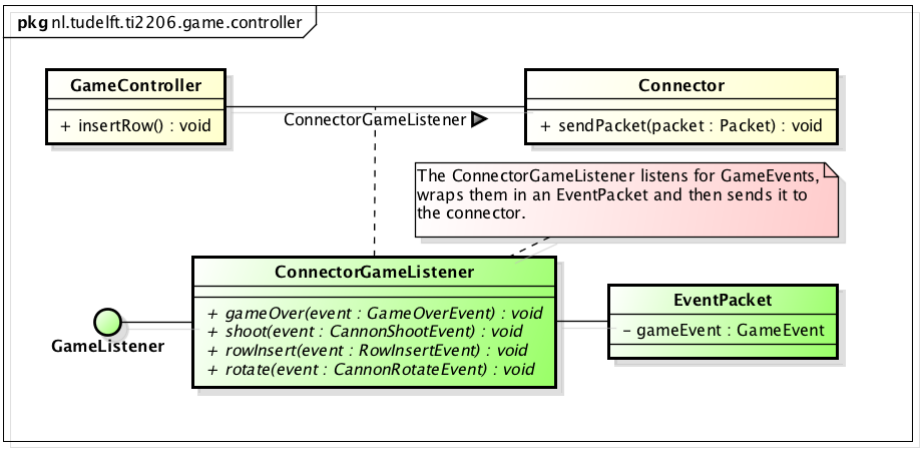

Figure 12: UML class diagram for the `ConnectorGameListener`

When the client receives a `EventPacket`, it needs to update the `GameController`. Therefore we have made a `PacketListener` that listens for `EventPackets`, and then invokes the event on the `GameController` (figure 13).

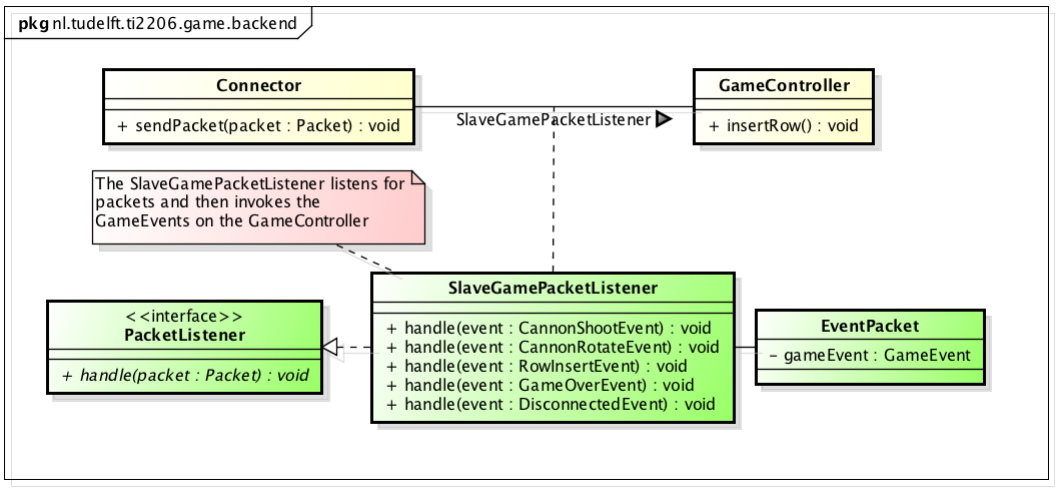

Figure 13: UML class diagram for the `SlaveGamePacketListener`

Now we have access to all required events in the multiplayer, which also allows the `GameModes` to work completely in multiplayer.

1.1.5 Pop animations

1.1.6 GUI Improvements

2 Wrap up - Reflection