Wallbreaker

-The Game-

## What is the main purpose of the program?

In the Wallbreaker Game the user plays with a bat to control a ball to tear down the wall. If the ball hits a brick, usually the brick will disappear, and the ball will bounce back. Since it is a game the main goal is to entertain the user.

The player will complete a level, when all the bricks (without the unbreakable) are broken.

After the user goes level to level, it will be harder and harder. The bricks wouldn’t break for the first time, and to complete the level the player has to be more accurate. There can be unbreakable bricks too.

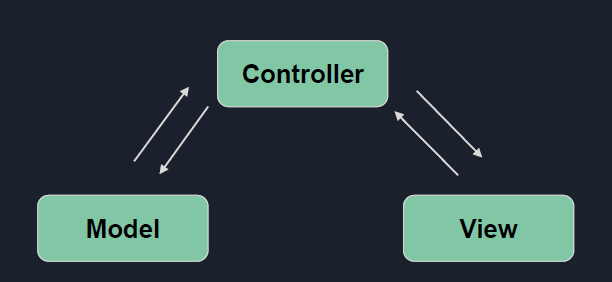
The player has limited life and if all balls fall down, the player loses a life. The ball only leaves the screen in the bottom, the ball will bounce back from the other three bound.

When a brick falls apart a bonus can fall. The user can decide to catch it or just leave it to fall down. The bonuses could be gifts, which make the game easier, or could be penalties, which make the game harder. Just a few bonuses: fire ball, bigger/smaller bat, the ball speed increase/decrease, bigger/smaller ball etc. Some of the bonuses will be temporary, so those will disappear after time. If all of the balls fall down, the new ball has no bonuses, but the bat can keep its bonuses.

## Structure, planning

For planning the program, we use paper and pen, but then, we made a UML file, to describe our plan.

When we thought about the bonuses, we had to make an easy way, to implement them, so we used the MVC model. It describes that the Model and the View cannot communicate each other directly, just through the Controller.



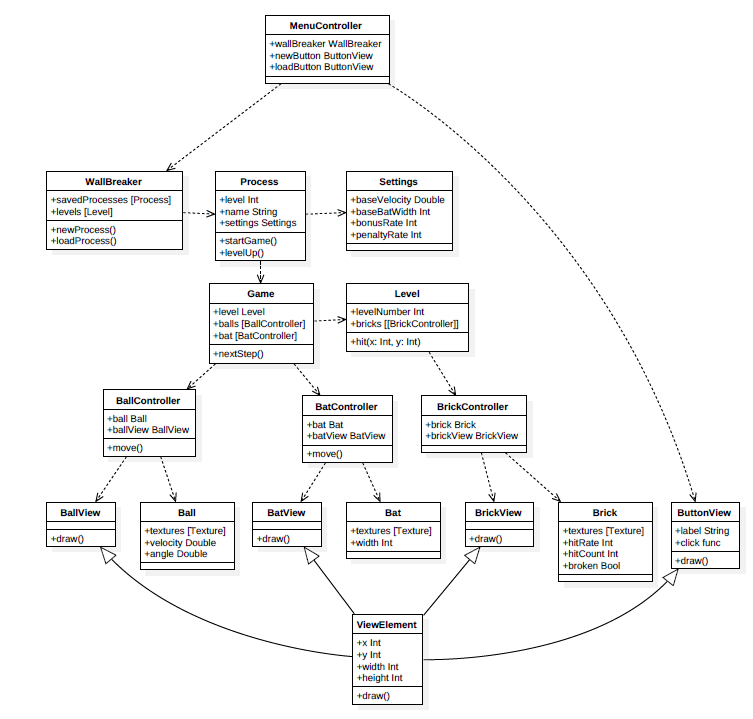
With this structure model, and with OOP, we could easily add new features, like “glued bat” or “fire-ball”.

Each entity has a model description and a view description. With the controllers we can make a ball or bat or brick, which has a description like what parameters it has, and a looking, like it is red or blue, in the screen where it is and so on.

To calculate with these data, we used the controllers, so we didn’t have to modify the original model.

For the bonuses’ case, we could put a new class, which inherits from the original. Eg.: if we want a glue bat, we can make a new class GlueBat, which inherits from the class Bat.

Here’s the planned UML file:



## Implementation

First, we had to implement the structure. We made a sceleton, with the planned functions.

We made a little research about what GUI implementer we want to use. This presented that python has a very good GUI, which is pygame.

**MenuController (.py)**

Pygame made our job easier. We made the Menu screen, where’s 3 buttons: New Game, Load Game, How to play. You can write your name (only 4 character) in there, so your saved game will be named. If you didn’t give your name, the savefile’s name will be None.

*Methods:*

*\_\_init\_\_() : for running the game*

*gui() : this contain the menu surface*

**Game**

With pygame, we made the gamespace, where we have walls, bricks, ball and the bat (and live, level).

*Methods:*

*refreshAllSpriteList (): this is needs to refresh the objects*

*looseLife(): this contain that when you lose one life in the game*

*initWalls(): this draw the play area*

*start(): this is the actual game mechanism*

**ViewElement**

We implemented these elements with sprites. Sprite is two dimensional image, a kind of object on the screen which interact with other elements. These were implemented in the ViewElements and all of them (except walls) load they appearance from file. These images drawn in Paint by Kiss Dorottya. The bricks have as many pictures as they hitrate is and these corresponding to the levels.

ViewElement is an abstract class. Inherit that:

*Buttonview, Ballview, Batview, Brickview: draw(): this contain the drawing of the objects*

**Level**

So, the level file contains the locations and actions of the bricks. The level’s layout is in a .csv file. There are only strings, which were processed by a function. If there’s no brick, its model will be None, and there won’t be Controller neighter View.

*Methods:*

*loadLevel(): load levels from file*

*loadBrick(): accordingly load bricks*

*deleteBrokenBricks(): delete broken bricks*

**Controllers**

When we had a simple GUI, we made the movements. The only items which had to move are the bat, and the ball.

In the case of the bat, we had to check the collision with the walls. Fortunately, in pygame, we had a built-in function pygame.sprite.collide. We only had to make a sprite group from the walls, and this function checks the collision. We wanted to stick the ball to the bat, so we made a boolean. So, in the beginning, the bat and the ball moves together, then (when the ball is released) separetly. The movement of the ball was solved according to the physical laws set by the setballreflect function.

*Methods:*

*Ballcontoroller:*

*Stickto(): the start of the game is from the jumper*

*move(), setBallReflect(), isOverlapped(): movement of the ball*

*isOnPlayField(): the ball go out of the play area*

*BatController:*

*changespeed(): change the bat speed*

*move(): movement of the bat*

*BrickController:*

*hit(): the ball hit the bricks*

**Process**

The backups are done automatically when you move to a new level. As I mentioned before the backups are done by name and these are stored in a map, so you can save up to several levels for a name.

*Methods:*

*startGame(): start a new game or start the loaded game*

*levelUp(): when you beat a level*

*saveLevel(): autosave to file*

## Difficulties

To calculate the movement of the ball, how to snap back and how to steer it (eventually it came to the left side of the racket, then snaps back to the left, right to right, centered to the right).

The game was lagging and as it turned out the reason was that the update() function of the bricks accidentally called the draw() function also so it ran two times in one cycle.