Game Screen Elements Subsystem

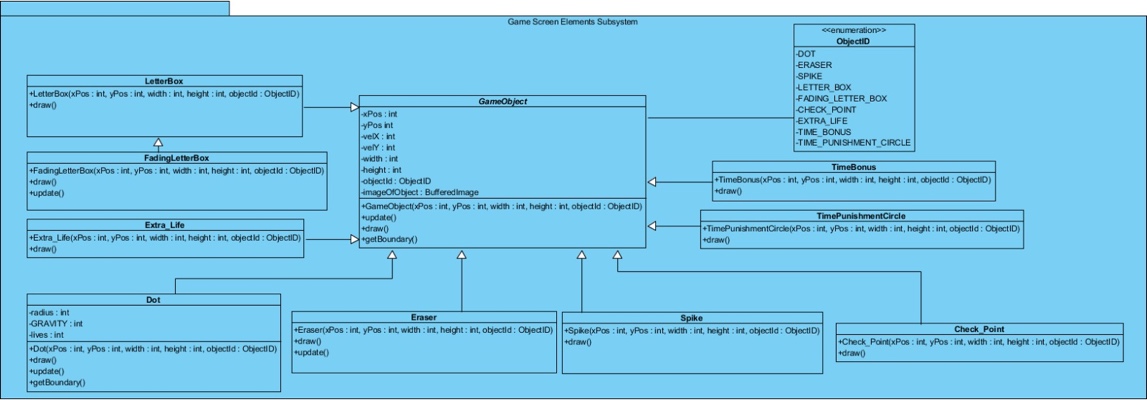
“Game Screen Elements Subsytem” plays a key role in declaring the objects which are shown in the screen while the game starts to running. There are many significant objects including “GameObject”, “LetterBox”, “FadingLetterBox”, “Extra\_Life”, “Dot”, “Eraser”, “Spike”, “CheckPoint”, “TimePunishmentCircle” and “TimeBonus”. This subsystem has also Enumaration named as “ObjectID” to indicate the types of different objects.

Figure 17 – Package Diagram of GameScreenElements

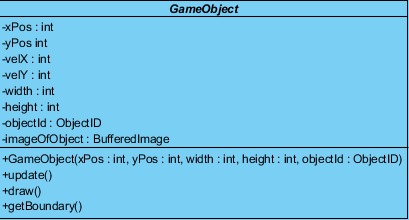
**GameObject Class**

Figure 18 - GameObject Class

“GameObject” is an essential class for this subsystem. All of the objects have to keep an account of the position information, velocity information, size information, object type and the image of object type hence, they use “GameObject” abstract class as a parent class which has all these attributes. This class stores the object type as “objectId” by using enumaration. Each object has a unique “objectId” defined as enum type. Also, there are abstract methods including “*update()”*, “*draw()”* and *“getBoundary()”* since all objects may act differently to these operations. “GameObject” will be instantiated whenever the user attempts to play the game.

A short description of the attributes and the abstract methods in the GameObject class to clarify the functionality of each attribute and method:

**Attributes:**

* **xPos :** The x coordinate of object’s position.
* **yPos :** The y coordinate of object’s position.
* **velX :** The velociy of object in the direction of x coordinate.
* **velY :** The velociy of object in the direction of y coordinate.
* **width :** The width of object.
* **height :** The height of object.
* **objectId :** The enum type of object.
* **imageOfObject :** This will stand for the images of different object types.

**Constructor:**

* **GameObject(xPos : int, yPos: int, width: int, height: int, objectId: ObjectID) :** This will initialize a game object according to given position values, size information and the object type.

**Methods:**

* **draw() :** This is an abstract method. All objects will include their own implementations but generally it will achieve drawing the object according to its size and position information. By calling this method, each game object can be drawn in its updated location.
* **update() :** This is also an abstract method. It will manage the position of object according to its updated position information. This method will get the x and y coordinate of object’s position and replace these values with the new coordination.
* **getBoundaries() :** This is also an abstract method and will be responsible for returning the boundaries of an object according to necessary attributes such as position, width and height of different object types.

***LetterBox* Class**

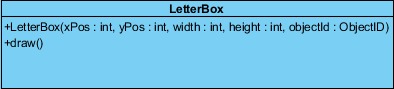


Figure 19 - LetterBox Class

The “LetterBox” class has the same attributes as the GameObject class but also includes a constructor for initializing LetterBox object and the implementation of *“draw()”* abstract method. This class will manage the game platform by drawing a number of letters and creating different sentences. The player has to jump over these letters without dropping between them. The LetterBox has a child class representing the fading letters which will be defined below.

The attributes of the LetterBox class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **LetterBox(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID ) :** This is the constructor where the position information, size information and the object type specification will be given, thus the LetterBox will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values. This constructor will initialize a number of letters which are constructing obstacles the player will be encountered during the game.

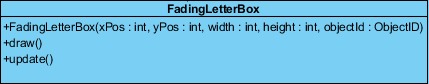
***FadingLetterBox* Class**

Figure 20 - FadingLetterBox Class

The “FadingLetterBox” class is a child class of “LetterBox” class and has the same attributes as its parent class but also includes the implementations of *“draw()”* and “*update()*” abstract methods. This class will be initialized after the player starts to play the level including fading letters. The FadingLetterBox object has the abilty to disappear when the collision with the Dot object is detected. Each object of this class is able to fade away in case of such a collision.

The attributes of the FadingLetterBox class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **FadingLetterBox(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the FadingLetterBox will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**Methods:**

* **update() :** This is an abstract method of the GameObject class. It will manage the position of object according to its updated position information. This method will get the x and y coordinate of the FadingLetterBox object’s position and change these values when the collision with dot object is detected and FadingLetterBox object needs to be disappeared.

***Extra\_Life* Class**

**design%20report/2017-12-02-PHOTO-00000428.jpg**

Figure 21 – Extra\_Life Class

The “Extra\_Life” class has the same attibutes as the GameObject class and the implementation of *“draw()”* abstract method exists in the class. The object of this class enables the player to get an extra life during the game. These “Extra\_Life” objects will be created around the letters and the player can manage to collect these extra lives which give the player one more chance to play.

The attributes of the Extra\_Life class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **Extra\_Life (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the extra life will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values. This constructor will create the extra lifes in specified positions where the player is able to reach and take it.

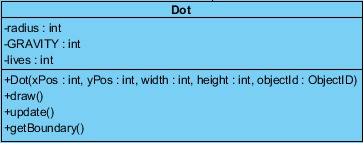
***Dot* Class**

Figure 22 - Dot Class

The “Dot” class has the same attibutes as the GameObject class but also includes a radius, GRAVITY and lives as new attributes. The implementations of *“draw()”,* “*update()*” and “*getBoundary()*” abstract methods exist in the class. The Dot object is as essential class for the game since it stands for the player who aims to finish the levels successfully. It will be created when the player starts to play first level.

The Dot class has the attributes of GameObject class which is defined before. Also, the description of abstract methods is given above. The new attributes of this class are described below to clarify the functionality of each attribute.

**Attributes:**

* **radius :** It represents the radius of dot object which will be drawn as a sphere. The draw method will use this attribute to specify the size of this object while drawing it.
* **GRAVITY :** This is the attribute which indicates the gravity of dot object because the dot is the only object that needs to be affected by the force of gravity while falling down. This object will jump between the letters without any collision with the ground then, it will be under influence of the gravity during the game.
* **lives :** This attribute keeps an account of the lives of the Dot. The Dot will initially have 3 lives. This information is given to the GameManager class to be able to show the remaining lives of the Dot in the user screen.

**Constructor:**

* **Dot (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the dot will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**Methods:**

* **getBoundary() :** This is an abstract method and will be responsible for returning the boundaries of the Dot object according to necessary attributes such as position, width and height of this object. Also, this method will help to check whether the dot is out of screen by calling this method and indicating the boundaries of dot object.

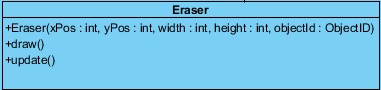
***Eraser* Class**

Figure 23 - Eraser Class

The “Eraser” class will be initialized when the player manages to play the level including the Eraser objects aimed at dropping from top of the screen and hitting the Dot object as player. Hence, the Eraser object will take a part in exposing the player to some obstacles during the game.

The attributes of the Eraser class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **Eraser(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the Eraser will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

***Spike* Class**

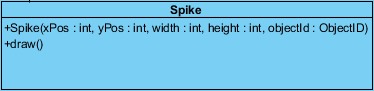


Figure 24 - Spike Class

The “Spike” class stands for another obstacle type during the game. It has the same attributes as the GameObject. The Spike object will be positioned according to its attributes including “xPos” and “yPos”.

The attributes of the Spike class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **Spike(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the spike will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**CheckPoint Class**

**design%20report/2017-12-02-PHOTO-00000425.jpg**

Figure 24 – CheckPoint Class

The “CheckPoint” class has the same attibutes as the GameObject class and the implementation of *“draw()”* abstract method. The objects of this class will be created in specified positions and these checkpoints can be reached by the player throughout each level. For instance, if the Dot falls out of the screen the life of the Dot is decreased by 1 and the Dot is placed to its starting position or to the last checkpoint that was reached. Hence, the checkpoint enables the player to finish each level without returning the starting point for its each fault.

The attributes of the CheckPoint class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **CheckPoint** **(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the CheckPoint will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**TimePunishmentCircle Class**

**design%20report/2017-12-02-PHOTO-00000435.jpg**

Figure 25 – TimePunishmentCircle Class

The “TimePunishmentCircle” has the same attibutes as the GameObject class and the implementation of *“draw()”* abstract method. The object of this class leads to a decrease in the remaining time of each level if the player collides with this object. Therefore, it stands for an obstacle for the game. The dot must try not to encounter a TimePunishmentCircle object.

The attributes of the TimePunishmentCircle class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **TimePunishmentCircle** **(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the TimePunishmentCircle will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**TimeBonus Class**

**design%20report/2017-12-02-PHOTO-00000434.jpg**

Figure 26 – TimeBonus Class

The “TimeBonus” has the same attibutes as the GameObject class and the implementation of *“draw()”* abstract method. This class provides a reverse situation for the player because TimeBonus object increases the remaining time of each level if the player encounters an object of this class.

The attributes of the TimeBonus class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

**Constructor:**

* **TimeBonus(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the TimeBonus will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

**Enumaration (ObjectID)**

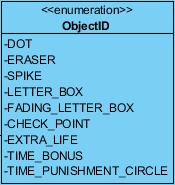


Figure 27 – ObjectID as Enum Types

“Game Screen Elements Subsytem” includes a lot of object types such as “Dot”, “Eraser”, “Spike”, “LetterBox” and “FadingLetterBox”, “CheckPoint”, “ExtraLife”, “TimeBonus” and “TimePunishmentCircle”. In attempt to make our implementation organization easier, we use enumaration types by specifying “ObjectID” related to each unique object. For instance, we can call “objectId.DOT” as objectId attribute of the Dot object when we define it in the “ObjectID” enumaration only once, since all objects use the GameObject as parent class having “objectID” attribute. With the advantage of enum types, the GameObject class can use different objects by calling only their unique “objectId”.