|  |
| --- |
| University OF THE WITWATERSRAND,  JOHANNESBURG |
| Centipede Game Design |
| ELEN3009 |
|  |
| **Sello Fotoyi 680492**  **Jerome Kabwayi 710392** |
| **07/10/2018** |

|  |
| --- |
| Abstract: Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. |

# Introduction

# Requirements

# Constraints

# Criteria for Success

# problem analysis

The original Centipede game is visualized as a system created by Atari. The inputs into and the output out of this system are the computer keyboard keys and screen, respectively. The project brief and the linked demonstration video clip present this system in the form of a black box to the developers and assigns to them a task of approximating the internal implementation of the system using the tools mentioned in sections 2 and 3. This problem analysis approach is illustrated in figure # below

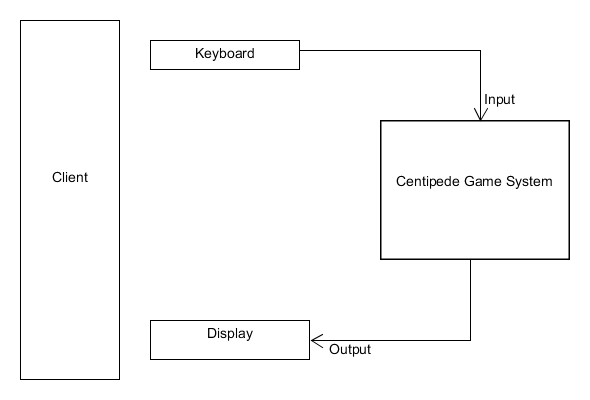


Figure #: problem visualization

The game system can be modelled by breaking it down into smaller subsystems, with each subsystem responsible for performing a specific task. A main subsystem could then be developed to bring about a logical game flow by providing a platform for interaction of these subsystems. This model idea coincides with the concept of object-orientated programming and is therefore used to solve the stated problem.

# design modelling

The domain of the system models Laser, Player, Field (of mushrooms) and Centipede as game objects visible to the user. Composition relationship is implemented to relate Laser to Player since lasers’ construction and existence occurs under player’s life cycle. Lasers exist only when a player “shoots”. Similarly, Mushroom objects are housed by Field object, and their existence also depends on the existence of Field object. However, Mushroom objects exist whenever Field object exists since mushrooms are constructed when field’s constructor is invoked. Centipede is conceptualized as a body with Segment objects as its building blocks, and thus Centipede and Segments’ relationship is also composition. Centipede guides the collective movement of these segments within screen borders and within a field of mushrooms.

Common properties shared by game objects, such as live state (alive or dead), position setting and retrieval, are captured by Entity object. The game objects relate to this invisibly internalized object through realization. Entity object in turn relates to yet another invisible Position object by realization as the Position object is solely responsible for position checking and setting of a game object. Moving objects use another internal object named Mover, which has the ability of changing a current position on the screen whenever instructed to do so. A moving object’s member function responsible for a moving action creates this Mover object and destroys it after using it. The objects’ construction, movement and collisions are discussed below

## Game Objects and their Construction

The validity-checking of objects occurs prior to their construction. An exception is thrown inside an object’s constructor, directly or indirectly, if desired conditions for construction are not met. For instance, an exception will be thrown if an object is attempted to be constructed with a position which is out of screen boundaries.

## Movement

Movement of an object (occurs in a rectilinear and movement speed is modelled by a float variable which is either added to or subtracted from a single position coordinate at a time to establish a new position.)*.* Checks are performed to prevent invalid object movements, such as movement beyond screen borders or after colliding with specific objects.

## Collisions

Game objects are modelled as Axis-aligned bounding boxes. An object-specific box is created from its position, and an algorithm is adopted to check if the boxes representing positions of different objects touch or overlap. Objects whose boxes have overlapped with specific objects’ boxes are removed from the screen

# design Structure Overview

## Presentation Layer

### Display Class

This class models the game’s screen by utilizing SFML RenderWindow class. It mimics basic RenderWindow member functions such as display(), clear() and close(). It uses SFML event class to check if the screen is closed by an escape key or window close icon. For construction, Display object requires SFML VideoMode class to be passed onto RenderWindow class’s constructor.

### SplashScreen Class

SplashScreen’s responsibility is to output the game’s name and instructions to the user when the game is initiated. A single member function is used to fulfil this responsibility. Internally, the class consists of helper functions with responsibilities ranging from setting the font for instructions’ string, to drawing the game name and instructions. The class uses SFML RenderWindow and Display class (discussed in section 6.1.1) to draw and display the game’s name and instructions.

### KeyReader Class

The class uses SFML Event and Keyboard classes to read keyboard inputs from the user. It specifically checks for buttons designated for player movement (arrow buttons), player shooting (space bar), game pause (P), and the Escape key, which is used to exit the game screen. KeyReader’s constructor requires SFML RenderWindow class, since the key-checking process occurs while a game screen is open. The class’s member function returns an enum type which signals the key pressed. This hides the SFML classes used for key-checking from KeyReader’s user and enforces layer separation.

### Drawer Class

As the name suggests, Drawer class draws game objects. Its constructor requires a shared pointer to Display object (discussed in 6.1.1). Drawer consists of member functions which are responsible for drawing specific game objects, such as Player and Field of Mushrooms. The member functions uses the objects’ EntityIDs to retrieve their corresponding SFML Sprites, and then uses the objects’ positions to draw on specific locations on the game screen.

## Logic Layer

### Position Class

The class is responsible for managing the positions of game objects on the game screen. This includes setting and retrieving their x and y coordinates. The class is similar to SFML Vector2D class and could have been used to perform the same tasks, but *Position* class is implemented with the sole purpose of attempting to separate layers, and detach the dependency of external libraries’ (such as SFML) classes in the logic layer.

### Entity Class

The classcaptures properties common to all game objects. It captures the position of a game object by a data member of type Position and returns it as a shared pointer to type Position whenever required. The class also has member functions which controls tell the live state and Entity ID of an object. Upon construction, the class’s constructor requires valid x and y coordinates of a game object and its corresponding Entity ID.

### Mover Class

Mover class instances are used to perform movement action of game objects, as described in section 5.2. This is performed using four member functions. A Mover instance will move a game object if it is within the set screen borders. It checks for this condition by making use of Boolean helper functions which will true if the position of an object is within screen borders. The class uses a Position object as its data member through composition, which will assist in setting and retrieving the position of an object after moving it. Mover’s constructor requires valid x and y coordinates of an object to be moved, and its corresponding speed.

### Laser Class

A laser object’s structure is modelled using an Entity concepted as discussed in section 6.2.2. This is done by importing Entity’s functionality by means of having it as a data member through composition, and provides access to Entity’s interface by returning a shared pointer to type Entity using a member function entityAttribute(). In addition, uses updatePosition() to traverse it vertically upwards. This member function destroys a Laser instance when it reaches the top of the screen.

### Player Class

This class represents the user in the game. It is also modelled using an Entity concept. In addition, as per project brief [#] and success criteria discussed in section 4, a Player object moves left, right and down within screen borders, can move upwards up to a certain set limit. The class uses Movement object to perform this task whenever its move member function is invoked. Player’s move function takes a Direction enum as its parameter to indicate the direction of movement. The enum type is handled through a switch statement.

A Player object is able to shoot lasers vertically upwards. This is achieved by having a container of Laser instances as its data member (composition). The lasers are constructed and pushed into a container(std::list) whenever Player’s shoot() is invoked. The class provides access to these lasers through its getLasers() member function. The lasers’ positions are updated by updateLasers(), which invokes updatePosition() of every Laser instance inside that container. Player’s construction requires its location on the screen (x,y coordinates), valid EntityID and a float variable representing speed.

### Mushroom Class

Mushroom’s basic structure is modelled by Entity class. A mushroom’s weaken() member function, is invoked externally whenever a mushroom collides with a laser. The member function uses an internal counter which keeps track of the number of times the mushroom has collided with a laser and changes its live state to false colliding with a laser four times.

### Field Class

Field class is responsible for generating and housing Mushroom instances. It models this container concept by a data member of a list of Mushroom objects by composition. The mushrooms’ positions are randomly generated by utilizing a random number generator. The Mushroom instances are created whenever a Field class constructor is invoked. The class has a single member function (getMushrooms()) which is of type Mushrooms(std::list<shared\_ptr<Mushroom>>), and is used to provide access to the Mushroom instances to be processed externally for collisions. The class’s constructor takes in a number of Mushroom objects to be generated.

### Segment Class

Segment class models a fraction of a centipede. Like Laser, Player and Mushroom classes, it is also modelled as having Entity properties. Segment class also uses Mover object to move, and also takes in a Direction enum as its move member function parameter. In addition, a Segment object knows where it is facing, such that its movement heads to the direction which it is facing. This feature uses internal direction flags. It is ensured that opposite directions are always complements of each other (a segment cannot face right and left or downwards and upwards at the same time). Segment class constructor requires (x,y) coordinates, valid EntityID and speed.

### Centipede Class

The class represents the main protagonist of the game. It is modelled as a container of Segment objects (std::list<shared\_ptr<Segment>>) through composition, and are constructed and pushed into a container when its constructor is invoked. Upon construction, the class uses a helper function to position the segments from top centre of the screen going left, and spaces them according to the size of the set segment width (Constants::SEGMENT\_WIDTH\_, see section 6.4.3), and The positions of the segments are located Its move() member function is invoked to perform its movement as described in sections 4 and 5.2.

Every Segment object is instructed to move down in a field of mushrooms as described in section #. The class uses a handleMushroom member function to make a Segment switch direction as it collides with a mushroom. This member function is called externally to check for collisions with the mushrooms, and it takes in a list of Mushroom objects to be collided with.

### Box Class

### (reference min-max box image)

The class is used to model the game objects as axis-aligned bounding boxes (AABBs). Its getBox member function takes in the (x,y) position of an object and its corresponding EntityID. It then uses a helper function to generate a conceptual box encapsulating a particular object. This is done by generating two positions, one directly below the given object’s position, and the other situated across the original point (to its right). This is illustrated in Figure #. The class is considered as the first step to collision handling, as the box created is used by a collision-detecting class.

### CollisionDetection Class

Upon construction, CollisionDetection class takes in the current positions and EntityIDs of the game objects to be tested for a possible collision. The class then employs the Box class to create a bounding box corresponding to each object’s position.

#include b2TestOverlap function from the Box2D engine

Its pairwise operation

### CollisionReaction Class

The main responsibility of this class is to erase collided objects from their corresponding containers. The class checks the live state of an object within a container (std::list) and erases it from that given list. Its updateLasers member function takes in a list of shared pointers to type Laser. The member function erases lasers which return false for Entity::isLive(). The same procedure follows for Mushroom objects passed in a list as a parameter for updateMushrooms member function. updateSegments member function takes in a list of Segments (Centipede) and a list of mushrooms (Field). A Mushroom object is constructed using the position of a segment which returns false for Entity::isLive() before that segment is erased from a list. The member functions use iterators to traverse through the forwarded lists.

## Data Layer

This layer consists of one class, namely DataBank. The class is used to retrive the image from memory, with its fractions used to depict game objects. The class makes use of SFML Sprite class to generate sprites and loads them whenever required. The class also retrieves and loads fonts used to depict the game’s name and playing instructions on the splash screen.

## Mixed Layer

### Enum Class

The class is used to store strongly-typed enumerations used to enforce layer separation as much as possible, while enabling interaction of such layers. For instance, enum class Pressed mimics SFML Keyboard class. This enum is used to alert a class in logic layer that a certain key was pressed, without having to pass it using SFML Keyboard.

### GameEngine Class

The class works as an intermediate connection of different layers and is responsible for a logical flow of game events (see section #). Its header file is included in client code (main.cpp), which has access to a single member function (playGame()). Invoking this member function invokes further helper functions to orchestrate game functionality in a logical manner. GameEngine can be visualized as a platform for interaction of different layers to bring about a working game.

### Constants Class

The class is used to store most common variables used in a number of classes. For instance, Constants::DISPLAY\_WIDTH\_ is used to set the width dimension of the screen in Display class. It is also used to restrict movement of moving objects to be within screen borders. This is achieved without having to include a presentation layer class in logic layer classes. This also ensures that information is hidden since the variables are not global.

# tests

## Construction

It is imperative to ensure that only valid versions of objects are constructed, and tests are written to confirm that validity prior to their construction. An instance of Position class is valid only if the x and y coordinates passed as parameters of its class constructor are not negative and are within designated screen borders. An exception is thrown if that validity condition is not met.

Since game objects utilizes a Position instance though the relationship described in section 6, validity of their positions are also tested, as an assurance valid positions are passed upon their construction. An exception is thrown in Position’s constructor if invalid positions are passed as parameters in game objects’ constructors.

In addition, game objects are marked as valid if a valid EntityID and speed (for moving game objects) are passed in their respective constructors. For instance, if an attempt to construct a player object with Mushroom’s EntityID and a speed other than Constants::PLAYER\_SPEED\_, respective exceptions are thrown in its constructor, which prevents the construction of that game object, and crashes the programme since the thrown exceptions are not handled[levitt lecture].

The validity of the number of segments and mushrooms to be generated are tested in by centipede and field instances respectively, are tested if they are greater than zero before the respective container class’s instances are constructed. Centipede also has a requirement for the maximum number of segments to be requested. A number larger than a specified maximum could result in an attempt to construct segments out of screen borders, marking those segments as invalid. An instance of CollisionDetection class is considered to be valid if collidable objects (as described in section #) are passed as parameters in its constructor. An exception is thrown if that case is not met.

## Movement

Each moving game object tests the validity of its movement functionality within, and if it restricted to, the designated screen borders. A special case occurs for centipede’s movement since it is supposed to either move down or up and then towards a direction opposite to a prior direction before arriving at a screen border. For contained moving objects, their movement functionality is tested again, but indirectly, through movement tests of container objects.

## Collisions

Section 72.11 describes an algorithm adopted for collision detection. The most extreme cases for collidable objects are tested. These are depicted in figure # below.

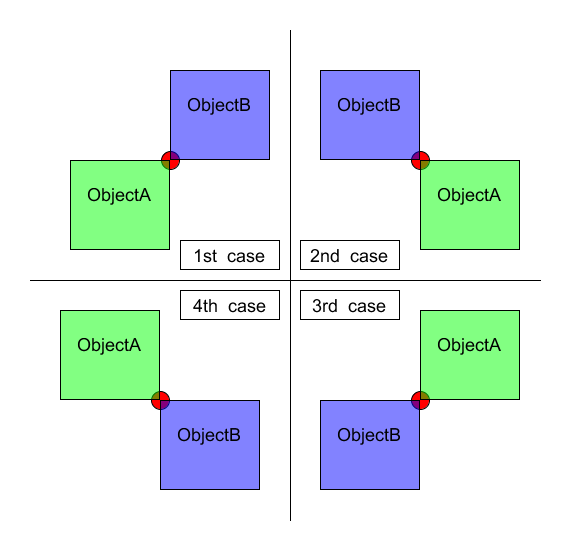


Figure # : collision-testing special cases

The cases in figure # are dubbed as being extreme since the algorithm adopted detects collision for overlapping objects, and that objects touching on corners are the least overlapping objects. CollisionReaction’s functionality is also included in tests, to check if objects destroyed due to collisions are erased from their respective containers.

## Untested logic

The relationship between game objects, Entity class and Position class is described in section 6. Game objects do not test the validity of all member functions of their type Entity class data member but one member function, which is the position getter. The logic behind that decision is that if position getter returns valid values, then all other member functions from this data member should behave as expected, since all these Entity’s member functions are tested under Entity tests.

Another untested logic is the collision detection for obviously overlapping objects. This is because since an algorithm adopted for collision detection is expected to detect those overlapping objects, and the special cases for collisions discussed in section 9.3 passes, then collision detection for obviously overlapping objects should be detected. Mover class’s functionality is not tested directly, but through movement tests of moving game objects. If moving objects can move, and are using Mover in that process, then Mover’s functionality is must be correctly implemented.

# design dynamics - object interactions

* #include sequence diagram / discuss

## Layer interaction

## Movement

## Collisions

# Critical Analysis

## Functionality Analysis

* Static comparison with solution
* Discuss centipede’s undefined mushroom collision, and their shrinkings
* Dynamic comparison with the solution

## Design Analysis

* Coding practices violated on the design
* Mention time complexities

# Future Improvements

## Design improvements

## Additional Features

# Conclusions

# References

|  |  |
| --- | --- |
| [1] | T. R. G. Geek, “Retro Review: Combat - Atari 2600 - 1977,” 18 February 2013. [Online]. Available: http://theretrogaminggeek.blogspot.com/2013/02/retro-review-combat-atari-2600-1977.html. [Accessed 3 October 2014]. |
| [2] | L. S.P., *ELEN 3009: Software Development II. Project 2014 - Tank Battle,* 1.0 ed., School of Electrical and Information Engineering, University of the Witwatersrand, 2014. |