**Project 3 Report**

**(PS. goodies may be referred as powerups or superpowers in places)**

**Actor class derived from GraphObject.h**

virtual void doSomething() = 0;

virtual bool BlocksMovement() { return false; }

bool isAlive() { return alive; }

void setDead() { alive = 0; }

StudentWorld\* getworld() { return world; }

virtual bool isDamagable() { return false; }

virtual bool isFlower() { return false; }

virtual bool isMushroom() { return false; }

virtual bool isStar() { return false; }

virtual bool f\_block() { return false; }

virtual bool m\_block() { return false; }

virtual bool s\_block() { return false; }

virtual bool isMovingInteraction() { return false; }

virtual bool isPeachFire() { return false; }

virtual bool isPiranhaFire() { return false; }

virtual bool hasGoodie() { return good; }

virtual bool isShell() { return false; }

virtual bool canShell() { return false; }

void noMoreGood() { good = 0; }

I have chosen doSomething to be purely virtual because every derived actor has its own implementation of this function.

BlocksMovement has been set to false and it only is true for objects that blocks movement of other actors. I have done this to accurately check if a particular tile blocks movement of any actor.

isAlive and setDead is used to fix if a particular actor is alive or not

getworld() returns the world from Studentworld.cpp

isDamagable() refers to the objects that can harm the peach ie all the enemies. This helps in checking the damaging of peach and damage done to enemies

isFlower, isMushroom, isStar checks if an object is a superpower respectively. This is used to determine which powerup is used in a particular case and accordingly assign the superpower.

f\_block, m\_block, s\_block are the blocks which have powerups (flower, mushroom and star respectively). Used for spawning powerups

isMovingInteraction is true for all the powerups used to collectively look at behavior of powerups

isPeachFire, isPiranhaFire is used to distinguish the two types of fireball and accordingly create its implementation

hasGoodie is used to check if a particular spot on the game has goodies which could be spawned. This ensured one special block only spawns one goodie. (good private member is set to 1 by default which gets removed by noMoreGood().) <- noMoreGood explained here

isShell refers to the shells which would move around the game killing other enemies and it a unique classifier for the shell class.

**BlocksMove class**

(derived from Actor) – this contains the objects which blocks movement

virtual void doSomething() { return; }

virtual bool BlocksMovement() { return true; }

* all the objects derived from Blocksmove do nothing in doSomething so it has been set a default in this base class.
* blocksMovement() has been set to true here as it is literally the meaning to this class.

**Block and Pipe** are derived from BlocksMove class and do not have any public function other than contructor

* f\_Block (flower holding block), s\_Block (star holding block), m\_Block (mushroom holding block) derived from BlocksMove class each have only public function ie f\_block(true) fro f\_Block, m\_block (true) for m\_Block and s\_block (true) for s\_Block

**Interaction class** derived from Actor does not have any public members but is only meant for storing Mario and flag under the same class for better organization.

Mario and Flag derived from Interaction only have doSomething()

* For a flag, level is completed and we proceed to the next level
* For a mario, game is over and plater wins

**class MovingInteraction**

is derived from Interaction class which is meant to store the goodies

void motion();

void switch\_dir();

void common\_doSomething();

virtual bool isMovingInteraction() { return true; }

virtual bool hasGoodie() { return true; }

motion() defines the motion of all the goodies as per the spec. It has been kept in this class because it is the same for all the goodies.

switch\_dir() is used in motion() to switch the direction between left and right according to the blocking condition.

common\_doSomething is further used in each goodie’s doSomething and holds the common actions needed to be done in each goodie’s case.

isMovingInteraction is the unique classifying Boolean value for this superclass and its subclasses.

hasGoodie is also a unique classifying bool which ensures that one spot releases only one goodie

**Flower class** derived from movinginteraction

* doSomething checks if it is overlapping with Peach, if it does it changes score, dies and gives the respective superpower. otherwise it moves according to the common motion function
* isFlower() returns true is a unique classifier for flowers used to give the respective superpowers.

**Mushroom class** derived from movinginteraction

* doSomething here does the same checking as Flower but the effects of interacting with peach is different (different power, different score increase)
* isMushroom() return true for identification purposes.

**Star class** derived from movingInteraction

* doSomething – similar to flower but different effects of peach interaction
* isStar() returns true for obvious reasons

**Fireball class** derived from Interaction (I know it can move but I have not kept in it under movingInteraction because I later decided to keep that class only for the goodies)

* f\_motion() is the common motion of all the fireballs (PS shell is also considered under fireball because it moves like one and damages other objects)

**Piranha Fireball** **class** derived from Fireball

* doSomething()- checks if it is alive, checks if it interacts with Peach and does the actions according to the spec (damage Peach and kill itself)
* isPiranhaFire() return true for unique identification

**Peach Fireball class** derived from Fireball

* doSomething() – checks if it is alive, checks if it hits an enemy and does the according actions (killing, increasing score and dying itself)
* isPeachFire() return true for unique identification

**Shell** **class** derived from Fireball

* doSomething() – chekcs if it alive, checks if it hits an enemy and does the according actions (killing, increasing score and dying itself)
* isShell() returns true for unique identification

**Peach class** derived from Actor

virtual void doSomething();

void block\_bonk();

void villain\_bonk();

virtual bool isDamagable() { return true; }

bool isinvincible() { return invinciblity > 0; }

bool istempinv() { return temp\_invinc > 0; }

bool canShoot() { return shootPower; }

void yesShoot() { shootPower = true; }

bool canJump() { return jumpPower; }

void yesJump() { jumpPower = true; }

void setInvinciblity(int n) { invinciblity = n; }

bool Starpower() { return hasstar; }

void yesStar() { hasstar = true; }

int howManyHits() { return hitpoint; }

void incH() { hitpoint++; }

doSomething() checks if its alive. it decrements the invincibility and temporary invincibility tick if its greater than 0. if it cannot fire right now then it would decrement the timer till it reaches zero. It then checks if it overlaps with other actors, if it does it acts accordingly. I have only written about contact with villain here because the other contact with peach cases are handled by other actor. If it has initiated a jump check it is it blocked by a wall or pipe above it. If it is blocked then cancel the jump. It then checks if it can FALL (the spot beneath it is empty).  
After checking all this, it takes input key that is pressed by the player and moves/fires accordingly.

block\_bonk() is called when peach hits a blockMove class member. It plays the sound and checks for goodie in the block

villain\_bonk() is called when peach hits an enemy. it either does damage to the peach or kills the enemy (if peach has star) or does nothing in case of temporary invincibility.

isDamagable() returns true because peach can be damaged. This is used in functions where damaging and killing takes place

isinvincible() checks if the peach is invincible ie has the effect of star goodie. This allows peach to kill enemies on contact if this returns true

istempinv() checks if peach is temporarily invincible. this does nothing in contact with an enemy of piranha fireball.

canShoot() returns if the peach has shootpower. This allows to use the Flower goodie accurately.

yesShoot() sets shootPower to true. this is used when peach contacts flower.

canJump() does the same for jumpPower (an effect of mushroom goodie)

yesJump() sets jumpPower to true. this is used when peach contacts mushroom

setInvinciblity() assignes n to invincibility. this is used when peach contacts star.

StarPower() does the same for invincibility ie effect of star goodie

yesStar() sets hasStar to true that means peach has contacted star goodie.

howManyHits() returns the hitpoint left in peach that can be used to check if its alive

incH() incrases the hitpoints (example when peach interacts with flower or mushroom)

**class Goomba** derived from Actor

virtual bool isDamagable() { return true; }

virtual void motion();

virtual void switch\_dir();

virtual void doSomething();

isDamagable() returns true because all the enemies can be damaged.

motion() implements the motion of Goomba

switch\_dir() is used by motion to switch the direction when Goomba hits a blockMovement object.

doSomething() check if its alive, otherwise it moves according to the motion() (its damaging is included in the attacker’s doSomething)

**class Koopa** derived from Actor

virtual bool isDamagable() { return true; }

virtual void motion();

virtual void switch\_dir();

virtual void doSomething();

virtual bool canShell() { return true; }

first 4 are exactly same as goomba (only difference is in doSomething of Koppa when it dies, a shell is formed which is not the case in goomba)

canShell() returns true because Koopa has the capability of making shell

**class Piranha** derived from Actor

virtual bool isDamagable() { return true; }

virtual void doSomething();

int givefireDelay() { return firing\_delay; }

isDamagable() returns true because all the enemies can be damaged.

doSomething() check if its alive, otherwise it faces the peach when peach comes within the given range and shoots fireball when it comes very close to it, changes firing delay and wait before firing delay reaches 0 again

giveFireDelay() returns the firing delay of the piranha

**class StudentWorld** derived from GameWorld

virtual int init();

virtual int move();

virtual void cleanUp();

~StudentWorld();

void level\_over() { lvl\_over = true; }

void game\_over() { g\_over = true; }

init() loads the game level and dynamically creates actors in the game.

move() is responsible for all the actors and peach to doSomething(). It checks if peach is not alive, decreases its lives. If level is over (when peach contacts flag), it plays the sound for finished level.

if game is over (when peach contacts Mario) sound is played and game is over.

move also deletes all the dead items in the game.

It is also responsible for diplaying the score, lives, level and points of the game on the top

cleanUp() function deletes all the dynamically allocated peach and actors of the game

~StudentWorld() just calls cleanup().

level\_over is used to indicate that the game is over thereby setting the lvl\_over private member true.

game\_over() is used to indicate if the game is over.

bool isBlockingAt(double x, double y);

bool overlapsPeach(Actor\* ac);

int PeachY();

int PeachX();

void p\_yesShoot();

void p\_yesJump();

void p\_yesStar();

void p\_setInvinc(int n);

void p\_incH();

void p\_villain\_bonk();

bool intersection(double x1, double y1, double x2, double y2);

bool intersection(Actor\* a, Actor\* b);

isBlockingAt() checks if there is a actor near the the parameter coordinates that has the ability to block movement (blocks and pipes)

overlapsPeach() checks if the actor ac is in contact with the peach. this is used in many functions in actor.cpp (one example when flag overlaps with peach it ends the level)

PeachY() and PeachX() are trivial functions which give the X and Y coordinates of Peach used for obtaining the position of peach in the game

p\_yesShoot() calls yesShoot() of peach that sets Shootpower true

p\_yesJump() calls yesJump() of peach that sets JumpPower true

p\_yesStar() calls yesStar() of peach that sets hasstar true

p\_setInvinc() calls setInvinciblity(n) of peach that sets invincibility to n ticks

p\_incH() calls inch() of peach that sets hitpoints of peach to 2.

p\_villain\_bonk() calls the villain bonk function of peach which perfroms the action of harm/damage to peach.

first intersection function checks if x1 x2 and y1 y2 are within the SPRITE range

and if it return true we say that x1,y1 and x2,y2 OVERLAP

the second intersection() checks if both actors OVERLAP by calling the first intersection

bool overlapFlower();

bool overlapMushroom();

bool overlapStar();

bool overlapVillain(Actor\* ac);

overlapFlower(), overlapMushroom(), overlapStar() checks if there is a overlap between the peach and flower/mushroom/star respectively

overlapVillain() checks if there is an interaction between the 3 villains and the actor ac (these ac are used in fireballs, shells, and peach). this is used to determine the state of villain after overlapping with a particular actor.

void checkGoodie(int x, int y);

void makepFire();

void makePiranhaFire(int x, int y, int dir);

void makeShell(int x, int y, int dir);

bool hasStar();

checkGoodie() checks if the particular location (according to the x, y) there is any goodie present. If yes, it spawns goodie in the location over it (as per the spec) and sets hasGoodie() in that location to 0 so no more goodie can be spawned in that location.

makepFire() simply creates a new Peach fire ball in the peach’s location and in peach’s direction

makePiranhaFire() creates a piranha fireball in the location and direction indicated. This is called when piranha detects that peach is close to it.

makeShell() creates a new shell in the location and direction indicated. It is called when a koopa dies.

hasStar() checks if peach has the star effect remaining. this is used to check if peach can kill villains by contacting.

void killPeachball(int x, int y);

void killPiranhaBall(int x, int y);

void killPower(int x, int y);

void killVillain();

void killVillain(Actor\* ac);

killPeachball() kills the peach fireball in the location indicated. this is done when peach ball either contacts an enemy or hits the block/pipe

killPiranhavall() kills the piranha fireball in the location indicated. this is done when it hits peach or block/pipe.

killPower() kills the goodie in that location. this is called when it hits peach.

killVillain() is called when peach comes in contact with villain. it kills the villain

killVillain(Actor\* ac) kills the enemy. It is called when Villain comes in contact with Peach fireball or shell or peach with star power.

**Functionality that failed and known bugs in my classes**

* I don’t think I missed anything from the demo game. If any, I think it would be any sort of undefined behavior in interaction of different actors because I kept getting that throughout this project. While submitting, I am confident that I have gotten rid of almost all of them
* There are a few bugs in my classes. I feel that there were many functions in class Actor which was either irrelevant or could have been coupled with other functions.
* Applying inheritance could have been better for the enemies as I copied a lot of code for them. Maybe I could have combined Mushroom and flower since their motion is quite similar and might have kept Piranha in a different category.
* There are a lot of functions in Studentworld.cpp which could have been avoided.
* There are a lot of overlap functions which could have been avoided with a better implementation of actor.cpp.

**Design Decisions and assumptions**

* Motions of all three goodies are same to its motion function has been included in the base class of all the goodies
* When contact of two objects involves bonking, damaging, killing of any sort, I have added all the subsequent actions in the class which is ATTACKING and kept nothing in the object BEING ATTACKED.
* For testing purposes I kept a flag at a easy reachable place to test different levels
* I have not kept different functions for bonking and damaging but kept the distinction in the one common bonk function only.
* I have created different class for special blocks ie the blocks which hold flower, star and mushroom which I feel is not the best method but it makes it easier to crease the appropriate powerup in the particular spot
* Creating goodie spawner took a lot of time. Also I saw in the game given to us that if we bonk a block from above and right left motion, it can spawn the goodie so I used a checkGoodie function in both above and left/right direction (based on the direction it is facing/moving)
* For assigning a goodie to peach I was having trouble doing it from Peach’s end (*when peach contacts goodie x, it should gain xyz power etc)* so I shifter all the responsibility to Goodies (*example when flower contacts Peach, it should killitself, increase score, assign shootpower to peach and give it a hitpoint).*