Cs 32 Project 3 report

1. Class Hierarchy

Class Hierarchy:

GraphObject

Actor

Peach

Block

Pipe

Piranha

Interactable Object

Flag

Mario

FlowerBlock

MushroomBlock

StarBlock

PowerUp

Flower

Mushroom

Star

MovingEnemy

Goomba

Koopa

MovingObject

PeachFireball

PirahnaFireball

Shell

These are the public member functions of the actor class:

Actor

* Actor(int ID, int x, int y, int direction, int depth, double size, bool dead, bool takesDamage, StudentWorld\* world);
  + The actor class has a constructor that includes the extra boolean parameters “dead” and “takesDamage” These parameters let me know when to remove an actor from the grid and actor behavior.
* bool isDamageable() const;
  + This function returns a boolean depending on the damageability of the actor. I chose to define it in the Actor class since all actors will need to define if they can take damage or not.
* bool isDead() const;
  + The function returns a boolean depending on if they have been set to dead or not. I chose to define this in the Actor class because it allows the StudentWorld program to detect which actors need to be removed.
* virtual bool isSolid() =0;
  + The function returns a boolean depending on whether or not other actors can pass through it. I chose to make this function pure virtual since in the game many actors have to redefine their behavior for isSolid and it was easier to make each class redefine their behavior than implement it as another parameter.
* StudentWorld\* getWorld() const;
  + The getWorld function returns the world, which will be needed for all StudentWorld function calls for other classes, so it was practical for it to be defined in Actor (which serves as my base class for all other classes).
* virtual void doSomething();
  + The doSomething function defines each actors behavior during a certain tick. I decided not to make this function pure virtual since other derived classes such as Pipe might have nothing under their doSomething functions. I decided to just declare this function as virtual and leave it empty to define the default behavior for a class that wouldn’t do anything during a tick.
* virtual bool isKoopa();
  + The isKoopa function defines behavior if the actor encountered is a Koopa. This function helps with my implementation of the shells and helps prevent memory errors when replacing a Koopa with a Shell. I decided to make this function virtual since the derived Koopa class would redefine the return value.
* virtual int getScoreIncrease();
  + The getScoreIncrease function helps with all derived classes that have specific score increases when they are removed from the grid. The function is declared as virtual since each derived class with a score increase would return a different value.
* void setScoreIncrease(int x)
  + The setScoreIncrease function helps with the initialization of actors with a specific score increase. Since this behavior is the same for all derived classes I just placed it in the Actor class.
* virtual void bonk();
  + I defined a bonk class that defines each actor’s behavior when another actor bumps into them. I decided not to define this function as pure virtual and simply left the function’s implementation empty because some classes may choose not to do anything when bonked and others would choose to redefine it. In the first case, this empty implementation would serve its purpose.
* void setDead();
  + The function setDead simply sets the actor class status to dead. Since most derived classes will die I decided to declare this as a function inside Actor.

Peach Class

Peach(int x, int y, StudentWorld\* world);

Constructor for the Peach class

* virtual void doSomething();
  + doSomething method that defines Peach’s movement and powerup behavior.
  + Function is virtual because other classes derived from Actor act differently during a tick.
* virtual bool isSolid()
  + isSolid method that defines Peach as a non-solid object.
  + Function is virtual because other classes might not be solid.
* void canShoot()
  + canShoot method defines whether or not Peach has the mushroom ability to shoot fireballs. The function is defined in Peach’s class because she is the only one who can get the flower as a powerup.
* void gotMushroom()
  + gotMushroom method defines whether or not the Peach object can have the 1.5x jump boost. The function is defined in peach because Peach is the only one who can get the mushroom powerup.
* void gotStar()
  + gotStar method behaves the same way as the other methods. Peach is the only one who can get the star so it is defined in peach.
* bool hasStar()
  + hasStar returns whether or not Peach has the star power. Defined in the peach class because only Peach can have powerups.
* bool hasMushroom()
  + hasMushroom returns whether or not Peach has the star power. Defined in the peach class because only Peach can have powerups.
* bool hasFlower()
  + hasFlower returns whether or not Peach has the star power. Defined in the peach class because only Peach can have powerups.
* void setInvTickCount(int ticks);
  + The setInvTickCount function defines how many ticks of invincibility Peach has. Since this function is accessed in the StudentWorld class I put it in Peach’s public methods.
  + Not defined as virtual since only Peach can go invincible
* int getInvTickCount() const;
  + The getInvTickCount function returns how many ticks of invincibility Peach has left. This function is accessed outside of Peach’s class so I defined it in the public functions.
  + Not defined as virtual since only Peach can go invincible
* bool isInvincible() const
  + isInvincible returns if the Peach object is currently invincible or not.
  + Not defined as virtual since only Peach can go invincible
* void decHp()
  + Since Peach can have 2 hp (health points), I defined a decHp function to decrement her Hp outside of the Peach class.
  + Not defined as virtual since only Peach can take damage and survive.
* void hpUp()
  + Since Peach increases to 2 health when picking up a powerup, I use this function to change her health to 2.
  + Not defined as virtual since only Peach can gain health
* int getHp() const;
  + Returns how much Hp Peach has left.
  + Not defined as virtual since only Peach can gain and lose health.
* void losePower();
  + Removes all status effects off of Peach and resets her health to 1.
  + Not defined as virtual since only Peach can pick up powerups

Pipe Class

* Pipe(int x, int y, StudentWorld\* world);
  + Constructor for pipe defines position and world
  + Constructors don’t need to be virtual
* virtual bool isSolid()
  + Pipe is a solid object which means no other actors can pass through it
  + Defined as virtual since it’s pure virtual in Actor class

Interactable Object Class

* InteractableObject(bool holdsGoodie, int ID, int x, int y, int direction, int depth, double size, bool dead, bool takesDamage, StudentWorld\* world) : Actor(ID, x, y,direction, depth, size, dead, takesDamage, world){};
  + Constructor doesn’t need to be virtual, declare an extra parameter for whether or not the block still holds a “goodie” inside it.
* bool hasGoodie()
  + hasGoodie function returns boolean whether or not the object still holds a goodie
  + Not virtual because all interactable objects (Flag and Mario) hold goodies.
* void useGoodie()
  + useGoodie makes it so the object no longer holds a goodie/powerup
  + Not virtual because all interactable objects use a goodie the same way.
* void initGoodie()
  + initGoodie is called when each interactable object is created and sets their hasGoodie status to true.
  + Not virtual because all interactable objects will initialize with a goodie the same way.

Flag Class

* Flag(bool holdsGoodie, int x, int y, StudentWorld\* world);
  + Constructor for Flag
* virtual bool isSolid()
  + Defines solid as false since flag is not solid
  + Virtual since other Interactable objects such as powerup blocks are not solid.
* virtual void bonk();
  + When flag is bonked it increases the players score
  + Virtual because other classes in Interactable object do different things when bonked
* virtual void doSomething();
  + doSomething checks if the flag overlaps with Peach and moves to the next level if so.
  + Virtual because other classes in Interactable object do different things during each tick

Mario Class

* Mario(bool holdsGoodie, int x, int y, StudentWorld\* world);
  + Constructor for Mario
  + Not virtual because it’s a constructor
* virtual bool isSolid()
  + Defines Mario as a non-solid
  + Virtual since other Interactable objects can be solid
* virtual void bonk();
  + Increments the player’s score by 1000
  + Virtual because other classes in Interactable object do different things when bonked
* virtual void doSomething();
  + doSomething checks if Mario overlaps with Peach and ends the game if so.
  + Virtual because other classes in Interactable object do different things during each tick

FlowerBlock Class

* FlowerBlock(bool holdsGoodie, int x, int y, StudentWorld\* world);
  + Constructor
* virtual bool isSolid()
  + Defines FlowerBlock as not solid
  + Virtual since other Interactable objects can be non-solid
* virtual void bonk();
  + Releases a flower powerup when it is bonked
  + Virtual because other classes in Interactable object do different things when bonked

MushroomBlock Class

* MushroomBlock(bool holdsGoodie, int x, int y, StudentWorld\* world);
  + Constructor
* virtual bool isSolid();
  + Defines MushroomBlock as not solid
  + Virtual since other Interactable objects can be non-solid
* virtual void bonk();
  + Releases a Mushroom powerup when it is bonked
  + Virtual because other classes in Interactable object do different things when bonked

StarBlock Class

* StarBlock(bool holdsGoodie, int x, int y, StudentWorld\* world);
  + Constructor
* virtual bool isSolid();
  + Defines StarBlock as not solid
  + Virtual since other Interactable objects can be non-solid
* virtual void bonk();
  + Releases a Star powerup when it is bonked
  + Virtual because other classes in Interactable object do different things when bonked

Powerup Class

* PowerUp(int ID, int x, int y, int direction, int depth, double size, bool dead, bool takesDamage, StudentWorld\* world) : Actor(ID, x, y,direction, depth, size, dead, takesDamage, world){};
  + Constructor for Power up parent class for all powerups: Flower, Mushroom, Star
* virtual bool isSolid();
  + Defines all power ups as non-solid
  + Virtual because Actor declares isSolid as a pure virtual method
* void doSomething();
  + doSomething() method defines the movement behavior for all power ups. It also defines what happens when Peach collides with a power up
  + Not virtual because all power ups move the same way and power Peach up when colliding.

Flower Class

* Flower(int x, int y, StudentWorld\* world);
  + Constructor

Mushroom Class

* Mushroom(int x, int y, StudentWorld\* world);
  + Constructor

Star Class

* Star(int x, int y, StudentWorld\* world);
  + Constructor

MovingEnemy Class

* MovingEnemy(int ID, int x, int y, int direction, int depth, double size, bool dead, bool takesDamage, StudentWorld\* world) : Actor(ID, x, y,direction, depth, size, dead, takesDamage, world){};
  + Constructor
  + Constructors are not virtual
* virtual bool isSolid();
  + All MovingEnemys are not solid so I define the pure virtual function from Actor as returning false;
* void doSomething();
  + All enemies (Koopas and Goombas) move at the same speed and turn around when encountering a wall. The doSomething function defines the movement and collision behavior with Peach and Peach fireballs.

Goomba Class

* Goomba(int x, int y, StudentWorld\* world);
  + Constructor

Koopa Class

* Koopa(int x, int y, StudentWorld\* world);
  + Constructor
* virtual bool isKoopa();
  + Returns true
  + This function is virtual because when checking for dead actors it knows to add a shell where it died.

Piranha Class (I spelled Piranha wrong in my code)

* Pirahna(int x, int y, StudentWorld\* world);
  + Piranha Class constructor
* virtual void doSomething();
  + doSomething function which checks if Peach is in range and fires if it doesn’t have a firing delay will fire a fireball. The piranha will also turn to face Peach if she is within a certain range.
* virtual bool isSolid()
  + Defines Piranhas as not solid.
  + Virtual function because the original function in Actor was a pure virtual function.
* void setFiringDelay()
  + Sets firing delay to a set amount (40) ticks so that Piranha
  + Not virtual since only Piranha has this firing delay
* void decFiringDelay()
  + Decrements firing delay every tick
  + Not virtual since only Piranha has this firing delay
* int getFiringDelay()
  + Returns the remaining firing delay
  + Not virtual since only Piranha has this firing delay

Moving Object Class (Parent class of Shell, Peach Fireball, and Piranha Fireball)

* MovingObject( int ID, int x, int y, int direction, int depth, double size, bool dead, bool takesDamage, StudentWorld\* world) : Actor(ID, x, y,direction, depth, size, dead, takesDamage, world){};
  + Constructor for MovingObject class
* virtual void doSomething();
  + All moving Objects will behave in very similar manners. The doSomething defines their movement and collision interaction behaviors and decides whether they can kill the actor they collide with or not.
  + Virtual because it redefines the empty Actor doSomething method.
* virtual bool isSolid()
  + All moving objects will be non-solid, isSolid returns false
  + Virtual because it defines the pure virtual function isSolid in Actor
* bool isKillsPeach()
  + Each moving object stores a boolean that determines whether or not it can kill Peach, this function returns that boolean
  + Not virtual because its functionality is used the same way for child classes
* void setKillsPeach(bool var)
  + Each moving object needs to set a boolean that determines whether or not it can kill Peach.
  + Not virtual because its functionality remains the same for all child classes.
* virtual bool isShell()
  + Each moving object class is not a shell by default, function returns false.
  + The function is virtual because in the Shell class it redefines the return value to true.

PeachFireball Class

* PeachFireball(int direction, int x, int y, StudentWorld\* world);
  + Constructor for Peach Fireball

PiranhaFireball Class

* PirahnaFireball(int direction, int x, int y, StudentWorld\* world);
  + Constructor for Piranha Fireball

Shell Class

* Shell(int direction, int x, int y, StudentWorld\* world);
  + Constructor
* virtual bool isShell();
  + Returns true since shell is a shell. Used to determine collision behavior.
  + Virtual since not all Moving Objects are shells.

StudentWorld Functions:

* StudentWorld(std::string assetPath)
  + Constructor.
* ~StudentWorld();
  + Destructor.
* void finishLevel();
  + Signals when to move on to the next level.
* virtual int init();
  + Initializes the game grid and all classes inside of the game.
* virtual int move();
  + Move is executed during each tick and moves all actors, clears dead actors, and displays score.
* virtual void cleanUp();
  + Clears the screen and destroys any pointers/dynamically allocated data
* bool canMove(int x, int y, bool ground);
  + Detects if an actor can move to a certain position without colliding with anything, and if ground is set to false, it will be allowed to bonk. Otherwise no bonking will occur. This parameter is to make sure that Peach will not bonk the ground below her when she lands after jumping (hence the name “ground”) but also works for when you don’t want a certain actor to be bonking.
* void addActor(Actor\* actor);
  + Adds an actor to the actors vector and increments the number of actors.
* bool overlap(Actor\* a);
  + Checks if an Actor “a” overlaps with the Peach actor.
* bool overlap(Actor\* a,Actor\* b);
  + Checks if two given actors a and b overlap with each other.
* void flowerPower();
  + Gives peach flower power.
* void addFlower(int x, int y);
  + Adds flower objects to the grid.
* void mushroomPower();
  + Gives peach the jump boost.
* void addMushroom(int x, int y);
  + Adds a mushroom actor to the grid.
* void starPower();
  + Gives Peach the star power.
* void addStar(int x, int y);
  + Adds a star to the grid.
* void takeDamage(Actor\* act);
  + General takeDamage function that sets actors to dead.
* bool hasStar();
  + Checks if peach currently holds a star.
* void peachTakeDmg();
  + Executes functions that are supposed to happen when Peach takes damage, removes powerups, sets invincibility frames, or dies.
* bool hitEnemyActor(Actor\* hitActor);
  + Detects if an actor has hit/overlapped with an Enemy actor (Koopa, Piranha, Goomba).
* bool pirahnaTargeting(int y);
  + Determines if Peach is in range to be fired at on Y axis.
* bool onRight(int x);
  + Determines which side of the Piranha Peach is on so that PIranha can turn.
* bool inRange(int x);
  + Determines if Peach is in firing range for the Piranha.
* void gameWon();
  + Determines if the game is over/won (Peach has reached Mario).
* void addShell(int x, int y, int direction);
  + Adds a Shell to the grid.
* bool overlapDamageable(Actor\* me);
  + Checks if Peach has overlapped with a damageable actor.

1. List of all functionality I failed to finish:
   1. None.
   2. BUG: My game ticks a tiny bit faster (difference of 1-2 ticks) than the actual game because of my computer but all my behavior matches the spec so that shouldn’t create any issues for people running the game other than me. On my computer, I had to change the MS\_PER\_FRAME to 6. Since this issue was mentioned in the spec as an ignorable issue.
2. Design assumptions I made:
   1. For my project, I pretty much followed the spec to the letter. One minor assumption I made was that I could redefine setDead to create different behavior when certain actors died. Specifically, when Koopa died I needed to spawn a shell in their place. Other than those I believe I followed the specifications exactly. I also assumed that the Shell dropped by Koopa randomly moving over empty spaces instead of falling 50% of the time was normal behavior since it occurred in the teacher-provided game too.
   2. I also assumed that the aforementioned tick difference in my game (where my game runs faster) was not a serious issue as mentioned in the FAQ.