Kimberly Nguyen, Max Mindel, Jason Fiammetta, Jerry Leung
Professor Mihajlovikj
CS3520: Programming in C++
March 23, 2018

Homework 4

Model

We have an interface called IWorld, which contains all of the world's objects. IWorld contains methods to initialize the world, advance the world on each tick, return the gamestate, and update the gamestate according to commands received from the controller. IWorld has instances of an ISoundListener and an IEffectListener, which listen for events that trigger any changes in sound (handled using the YSE audio library) or visual effects, respectively. YseSoundListener implements ISoundListener and contains methods to start or stop the music or to play a sound effect corresponding to specific key presses, such as if the player jumps. SDLView, described below, implements IEffectListener and contains a method to render an effect at a specific position. For example, it could be used to render sparks at the player's shoes.

World implements IWorld and contains a map of GameObjs. GameObj is an abstract class, extended by the classes Player and Platform. GameObjects are created using Box2D, which is a 2D physics engine. To encapsulate the physics logic in the model, the WorldState that World returns when the controller asks for the gamestate is simply a vector of ObjState structs, which contain information about the objects to be rendered. More specifically, an ObjState struct contains an image path for the object to be rendered and the position, represented as a Posn struct, at which that image should be rendered. This way, the controller does not have to know what a GameObj is.

View

We have an interface called IView, which has an instance of an IKeyListener. IKeyListener contains a method to handle key presses. IView contains methods to initialize the window and render the gamestate. SDLView implements IView and contains a TextureManager, which is needed to render objects in a window when using the SDL library. TextureManager would contain fields and methods specific to using this library. SDLView also implements IEffectListener so that it can respond to non-object-based rendering requests from the IWorld.

Controller

We have a GameController class, which has instances of an IWorld and an IView. GameController takes input from the view and passes it to its IWorld instance. This instance would then return the updated gamestate, which GameController then passes to its IView instance. GameController also implements IKeyListener, as it must be able to accept and

interpret the key commands registered by the view. This is all done in a loop that runs until the game is closed.

Relationships/Information

Our UML diagram provides a visual of the relationships between the objects and the information stored by them, which are also explained in the sections above. The table below provides a more concise view of the information stored in each object (if applicable):

*Interfaces are bolded; abstract classes are italicized.

Objects	**Fields**
IWorld	- ISoundListener sl // listens for sound triggers - IEffectListener el // listens for effect triggers
World	- map<GameObj> objs // all of the game objects to render
GameObj	- b2Body body // needed for Box2D
Player	- b2Vec2 jumpImpulse // amount of force to apply to the jump - b2Vec2 leftForce // amount of force to apply to the left - b2Vec2 rightForce // amount of force to apply to the right
WorldState	+ vector<ObjState> states // info about the game objects
ObjState	+ string imgSrc // file path to the image to render + Posn coordinates // where to render the image
Posn	+ int xpos // x-coordinate + int ypos // y-coordinate
GameController	- IWorld world // world instance (model) - IView view // view instance
IView	- IKeyListener kl // listens for key presses
SDLView	- TextureManager tex // used to render objects onto a window

Operations

Our UML also provides a visual of the operations provided by each object, which are explained in the sections above as well. The table below provides a more concise view of the operations provided by each object (if applicable):

*Interfaces are bolded; abstract classes are italicized.

Objects	**Operations**

IWorld	+ void initialize() // creates the world
	+ void tick() // advances the world by a tick
	+ WorldState getState() // returns the gamestate
	+ void command(int command) // updates the state based on the key
	+ static const (int for each command) // identifies each command

World	+ World(SoundListener sl, EffectListener el) // constructor
GameObj	+ b2Body getBody() // returns the Box2D body

Player	+ Player(b2world world, b2Vec2 startPosn) // constructor
	+ boolean jump() // returns true if the player has jumped
	+ void pushLeft() // moves the player to the left
	+ void pushRight() // moves the player to the right

Platform	+ Platform(b2world world, b2Vec2 startPosn) // constructor

ISoundListener	+ void startMusic() // starts the background music
	+ void stopMusic() // stops the background music
	+ void playEffect(string command) // plays a sound effect for the key press

IEffectListener	+ void acceptEffect(string command, int x, int y) // renders an effect

GameController	+ void initialize() // initializes the game
	+ void run() // starts the main game loop
	+ void listen() // listens for events

IKeyListener	+ void handleKey(int key) // handles a key-command + static const (int for each command) // identifies each key-command
IView	+ void initialize(int w, int h, IKeyListener kl) // initializes the view + void render(WorldState state) // renders the gamestate

Data Structures

We are using a map to hold the GameObjs in World. Each object has a unique ID, so the keys would be the IDs and the values would be the image addresses corresponding to the object. This would allow O(1) constant lookups, which is efficient for scenarios in which we would want to change the image of an object mid-game. However, we are using a vector to hold the ObjStates because we would want to render every object that is returned when getState() is called. Thus, lookup speed does not matter because we would be iterating through the entire vector each time. This can be done in O(n) (linear) time.