Revised API: Scrolling Platformer Team

Team Information

Scrolling Platformer Team:

- Scott Valentine (sdv4)
- Ross Cahoon (*rpc8*)
- Danny Goodman (*pdg7*)
- Deo Fagnisse (*df56*)
- Jay Wang (*jmw86*)

Division of Labor

- Scott Valentine - Animation, Statistics, Sprite State, level / level_management, input
- Ross Cahoon - ScrollingManager, Level, ScrollingGame, Player
- Danny Goodman - level_editor, level_management, ScrollingGame, Level
- Deo Fagnisse - LevelEditor, LEView, All of the LevelEditor package
- Jay Wang - collision_manager, sprite superclasses, sprite interfaces, movement

---

How to make a scrolling platformer game

Instructions for Game Building

1. Create a directory for the game with the game's name.
2. Inside this game directory, create a directory for all the levels named "levels" as well as an image directory named "images". Any images that are being planned on being used should be in this directory. Within the images package, place any backgrounds in a sub-package called images.backgrounds.
3. In the game directory create a Sprite Library java file that implements ISpriteLibrary, you will implement types of Sprites that will be in you game in this file with the exception of the Sprite that extends Player. All Sprite classes in this file will extend either **NonStaticEntity** or **StaticEntity** (found in vooga.scroller.sprites.superclasses). All behavior not associated with Animations or Collisions with other sprites (Ex. Gravity, Friction) will be described in these classes. The Animations and Collisions
will be described in their own file that is also located in the same directory as this file.
4. To handle collisions, you need to create a **VisitMethods.java** file that details exactly (1) what collisions you want to handle and (2) how you want to handle those

collisions. Please reference the current VisitMethods.java for an example of how this works. Remember, you can define a collision between two Sprites (i.e. Mario/Platform collision) or you can define a collision between two Interfaces (i.e. IPlayer/IEnemy). The **MarioCollisions.java** is a just a file that holds helper functions that I may want to call from VisitMethods. For example, I keep my marioAndPlatform collision logic in MarioCollisions.java. All your game specific collision implementation can be placed there. If you would like more collision granularity, use the **CollisionDirection.java** class. That class has a method called getCollisionDirection that takes two Sprites: sprite1 and sprite2. It returns the direction sprite1 collides with sprite2 in.

5. In the game directory, create a java file that extends Player.java, found in vooga.scroller.sprites.superclasses. All behavior such as input, animations, collision handling, as well as state variables will be held in here. Animations and Collisions will be described in their own file, just like the Sprites in the library.
6. Create Levels with the Level Editor. To run the LevelEditor, create a main class in the game directory that contains the public static void main string args method. Within this method call the static method runLevelEditor in your game class. The parameters for this method are the SpriteLibrary that you created in step 3, and an array of filenames of the background images that you want to use for your levels. These background images should be located in the images.backgrounds package as stated in step 2.
7. Levels are created by clicking the radio buttons on the side to select the sprite and clicking a box in the grid to the left to place the selected sprite. Right click to delete a sprite. A level can be saved, loaded, and simulated from the MenuBar. When saving the level, save in the levels package in your game directory because this is the location that the game will look when compiling your created levels. Multiple workspaces can be open at once to create multiple levels. Close the Level Editor when finished.
8. In the game directory, create a java file that extends ScrollingGame.java, found in vooga.scroller.model. In this file you will be required to implement several methods:
    - setPlayer() - You will instantiate the Player you have created earlier here and make this method return that instantiation.
    - setScrollingManager() - You will instantiate the ScrollingManager your game will be using. The currently implemented ScrollingManagers include UniScrollingManager and OmniScrollingManager. You can learn more about these in the API documentation.
    - setLevelFileNames()- You will return the names of the levels you have created in the LevelEditor in the order you would like them to be played in a String array.

- setTitle() - Set the title of your game that will be displayed on the Window.

Optional Game Features
These are features that are not required to build a game but can add more detail.

Animation
1. Write all AnimationState extending classes with appropriate isValid() methods and set their images. These dictate what the animation will display given the current state of the sprite.
2. Instantiate an animation and add all of the AnimationStates to the animation. This creates a working animation separate from a specific sprite.
3. Set the view of a sprite as the animation. This sets the specified sprite to this animation and allows painting the sprite to display the images of the animation. **NOTE:** currently, an animation can only be a component of a single sprite. You cannot pass the same animation to different sprites.
4. For an examples see sprites.animation.MoveLeft.java or sprites.animation.MovingSpriteAnimationFactory.java.

Statistics
This provides an interface for classes that deal with data in a systematic way. Implementing this can allow the developer to keep track of player stats (such as points or money) and even (in more sophisticated implementations) player inventory.
1. Write a class the implements Statistics and give an instance of this object to your player (such as a score).
2. Define actions that either add, remove, or get aggregate score of this object. In the case of score, the intersection of coins and the player result in player.getScore().addValue(100).

Sprite State
This provides a way to define different states for sprite. Examples include making the sprites invisible, giving the sprites super powers (such as super speed and killer attacks), or giving the sprites special abilities (such as the ability to throw fireballs).
1. Write class that extends State and pass this to the state manager of the desired sprite to add it to. Sprite states can work on multiple sprites at the same time, so for something like InvisibleState, only one instance can be used for every sprite that has the ability to become invisible.
2. For examples see sprites.state.InvisibleState.java

Design Goals and Descriptions

1) Support several types of side scrolling video games. These types of scrolling will in include a uni-directional scroller (Mario) and omni-directional scroller. These options should be easily

configurable to provide flexibility with the type of scroller is being created. Support for the scrolling methods will be module that is encapsulated in ScrollingManager. The Scrolling Manager can be extended if these two types of scrollers are not suffice for scrolling game needs.

2) All objects that are not merely images in our framework will be of some type of Sprite. A hierarchy of sprites will be created and is described below, under primary classes. The point of the Sprite hierarchy is provide function and flexibility to what is added to the game. From the hierarchy the game designer will extend a Sprite that meets his/her criteria. If further customization of the Sprite is needed, the State and Visitor design patterns will be used to change the behavior (also described below)

3) Drag and drop level editor. For the first iteration, the editor will have a variety of sprites that can be implemented in the game from a menu selection. There will be a workspace that symbolizes a level in which you can drag and drop the sprites to create a layout of the level. This will allow the designer to make the level by purely mouse actions and when the the level is done, the Level Editor will allow the user to save the Level with all it's characteristics to be used in the framework. In a second iteration, it is our hope to expand on customizing and saving Sprites that are entered into the Level, instead of using preset values. A LevelEditor will be accessed through a Game. When a Level is created, it will be saved as a serializable object to later be read when the game designer is making source code.

4) Support the Input group's API for all input. With this we will use their framework so we will have access to a rich library of inputs for our game designers.

5) Rules and Goals will be inherently included in the behavior of a Sprite. The Sprite will enforce rules when collisions occur and when it is updating. The goal of the Level is to get to a Sprite which could take you to the next Level. The list of Sprites within Model will begin with a StartLevel (extension of Level) that is our start menu then is preceded by a Linked List of Levels which ultimately is linked to a EndLevel (also extends Level) which is equal to winning the game.

Primary Classes and Methods

LevelEditor

- **Interfaces:**
    - ILevelEditor - processCommand(Editable,String)
    - IView - render(Renderable), processCommand(String)
    - Editable - addNewSprite(Sprite), deleteSprite(Location), changeBackground()
    - Renderable - paint(Graphics2D), getState()
    - IWindow - addWorkspace(WorkspaceView,Renderable)
- **LEController**
    - contains LEView through IWindow interface.

    - contains LevelEditor through ILevelEditor interface
    - Maps WorkspaceView LEWorkspaceView to its corresponding Editable LEGrid
- **LEView:** implements IView and IWindow
    - contains LEController
    - talks to LEWorkspaceView through LEController
- **LEWorkspaceView:** implements IView
    - contains LEView through IView interface
    - contains LevelView through IView interface
    - contains EditorView through IView interface
- **LEGridView:** implements IView
    - contains LEWorkspaceView through IView interface
    - renders Renderable LEGrid
- **LEToolsView:** implements IView
    - talks to LEWorkspaceView through IView interface
- **LevelEditor:** implements ILevelEditor
    - contains LEGrid through Editable interface
- **LEGrid:** implements Renderable and Editable
- **LETools:** responsible for setting the map of sprite radiobuttons to sprite IDs.

- **IView** *interface*
    - public void processCommand(String command)
        - //Pass a string that will be parsed and detected from a small command library in the LevelEditor in order to process the information from the LEView
    - This design choice allows extensibility insofar that it is able to handle an array of View actions; it does not limit the number of commands on the LevelEditor side
    - Example: ILevelEditor.processCommand("createsprite 1 500 600")

Our LevelEditor API utilizes a framework from Deo and Ross's SLogo group that incorporates the abstract classes Window, WindowComponent, and WorkspaceView. The Window is the frame of the LevelEditor and can contain multiple LEWorkspaceViews. All components that can be added to the Window are WindowComponents. These WorkspaceViews represent editing multiple Levels at once. The LEController maps the WorkspaceViews to their corresponding Editable. The LEWorkspaceView contains two components: LEGridView and LEToolsView. The LEGridView is responsible for rendering the LEGrid object associated with the workspace. LEGrid implements both Editable and Renderable. The LEToolsView will contain sprites, backgrounds, and other settings to choose from when creating the Level. The LEToolsView is populated from its constructor and does not change throughout the program.

The LEGrid object contains SpriteBoxes that can contain one sprite at a time. The SpriteBox and LEGrid classes allow a level of quantization to the LevelEditor such that the size of a building

block for the LevelEditor can be specified. When the Level is ready for simulation or saving, the LEGrid creates a Level with the sprites it contains.

The IView interface is implemented by all WindowComponents. This allows the LEGridView to pass a command through its parent components, through the LEController, to the LevelEditor. IView contains processCommand(String) and render(Renderable). This means that all components can pass the command up through the containers and pass the Renderable down through the containers.

Sprites

- Player Sprite

- Static Entity Sprites
    - Handled through an abstract superclass called **StaticEntity.java**
    - As the name suggests, this is the class you extend for your sprites that do not move
    - Examples include:
        - Plant
        - Platform
        - Coin

- Non-Static Entity Sprites
    - Handled through an abstract superclass called **NonStaticEntity.java**
    - Main focus of this class is to assist make sprite creation of moving sprites as simply as possible
    - Examples of Sprites that would extend NonStaticEntity:
        - Koopa
        - Moving Platforms
    - There is one abstract method that needs to be created in any subclass of NonStaticEntity - getMovement
        - `public abstract Vector getMovement(Movement movement);`
    - There are other helper methods that pertain to velocity
        - `public void changeVelocity(Vector vector)`
        - `public Vector getRandomVelocity()`

- Sprite Interfaces
    - The main purpose of these interfaces is to help consolidate the number of visit methods needed

- Before, visit methods operated on a sprite level (i.e. Mario/Platform)
- However, this seemed grossly unnecessary and led to much duplicated code
- For example, a Mario/Koopa interaction is very similar to a Mario/Alien interaction or a Mario/Turtle
- Ultimately, those are all collisions between the player and an enemy
- Thus, we have decided to group our sprites by interfaces, which led to a much smaller number of visit methods needed
- **Interfaces we have implemented:** IPlayer, ICollectible, IPlatform, IEnemy, ILevelPortal, ISprite
- Sample Interface: *ICollectible*

```
public interface ICollectible extends ISprite {
    public int getValue();
    public void takeHit(int hitValue);
    public int getHealth();
}
```

- Anything methods that the visit methods need to handle a collision between two spirtse needs to be implemented in the sprites' respective interfaces

Collision Management

- Collision Manager (CollisionManager.java)
  - This handles most of the collision handling implementation at the highest level.
  - `public void handleCollision (Sprite sprite1, Sprite sprite2)`
    - This method takes two sprites and it looks for the visit method that handles the specific collision we are looking for
    - First, it will look for the visit method on an Interface level (i.e. IPlayer/IEnemy collision)
    - If it cannot find a visit method on an Interface level, it will look for a visit method on a Sprite level (i.e. Mario/Koopa collision)
    - If it cannot find any visit method, CollisionManager will treat that collision as one where nothing is supposed to happen
  - `public void invokeVisit (@SuppressWarnings("rawtypes") Class[] classArray, Object[] sprites)`
    - Once we have found the correct visit method, we will invoke it on the sprites

- Collision Direction (CollisionDirection.java)
  - This is an entity that lets you know the direction two sprites have collided in

(i.e. if Mario collided with Platform from the TOP direction)

- `Direction collisionDirection (ISprite sprite1, ISprite sprite2) {`
    - This method will return a Direction (which is an enum)
        - It will either be TOP, BOTTOM, LEFT, or RIGHT

- Visit Methods (VisitMethods.java)
    - This is where you want to implement all your visit methods
    - Note - you do not need to implement a visit method for every possible type of collision
    - If you don't want anything to happen when two sprites collide, don't write a visit method for that collision
    - Sample Visit Method:

```
public void visit (IPlayer player, ICollectible coin) {
    player.incrementScore(coin.getValue());
    coin.takeHit(player.getHit());
}
```

Sprite Features (i.e. Statistics, Animation, or State)

- Sprite State (State.java)
    - This interface defines methods that a sprite can use to change how it reacts with its environment in different ways. It defines how the given sprite that uses it knows when to use it, and how to represent it in terms of sprite behavior (through `update`) and how to represent it visually to the user (through `paint`).
    - `public void update (double elapsedTime, Dimension bounds);`
        - Defines how this state updates the sprite. For example, the State FastState.java makes the player move twices as fast as normal by translating the sprite at twices its given velocity.
    - `public void activate ();`
        - This method activates this state for the current sprite.
    - `public void deactivate();`
        - This deactivates the state for the current sprite. If no other state is active, this returns to the default state of the sprite.
    - `public void paint(Graphics2D pen);`
        - This defines how the state is painted. For example, consider the state Invisible, which turns the player invisible to the user. This method

would leave paint(..) empty since not painting the sprite will make it appear invisible on the screen.

- StateManager.java
  - This class acts as a manager for all the possible states that a sprite might need to implement. Essentially, all sprites will start with a State Manager and initialize all of the states in the StateManager by adding them via `addState(int stateID, state)` method. A few things to note: all states are mapped using a stateID. This is how the sprite controls which state it currently uses. This is done through the `changeState(int stateID)` method, which switches the current state to the specified mapped state.

- Statistics
  - This interface is used to set standard methods in order to collect and keep track of player data and statistics.
  - `public void addValue(int val);`
    - Adds the value the this statistic's data
  - `public void removeValue(int val);`
    - Removes the value from this statistic's data.
  - `public int getAggregateValue();`
    - Gives an aggregate value for statistic. This could be a sum (in the case of score) or the size of the data set (for the implementation of a user inventory).
  - `public String getName();`
    - Gives the name of this statistic. This is often used for displaying purposes. For example, Score would return the name “Score”, while in a player inventory, it might return the name of the current item.

- Animation
  - This class is used to give animation to sprite. It allows game developers to define a image manager for a given sprite such that the image displayed depends on the current state of the given sprite. For example, this can be used to create an Animation with the AnimationStates of MoveLeft and MoveRight which display different images when the sprite moves left and when the sprite moves right respectively.
  - AnimationState.java
    - These are the classes that the game developper must define. To do this, the game dev extends this class and implements the required

methods. This uses a isValid(Sprite) method to determine whether or not to use this animation state as the current image of the sprite.
    - isValid(Sprite)
        - used for determining whether this animation state is valid
        - ex: for moveLeft isValid returns the boolean sprite.getX() sprite.getLastX() < 0
    - getView()
        - returns the Pixmap that is painted when this AnimationState's isValid == true.

- Movement
    - This is an abstract superclass that handles all movement for a NonStaticEntity
    - If you want a specific type of Movement (AI) for your sprite, extending NonStaticEntity will force you to implement a method getMovement(Movement movement) which returns a calculated Vector for your sprite to translate
    - You will need to provide what type of movement you want
    - Note that every movement subclass you create needs to implement execute (which returns a Vector)
    - We have made some sample movements
        - i.e. TrackPlayers
        - ```
          public Vector execute (int speed, int radius, Player myPlayer)
          ```
        - ```
          Location player = myPlayer.getCenter();
          ```
        - ```
          if (Vector.distanceBetween(player, myEntity.getCenter()) > (double) radius) return NonStaticEntity.DEFAULT_SPEED;
          ```
        - ```
          return new Vector(Vector.angleBetween(player, myEntity.getCenter()), speed);
          ```

Level

- Level.java
    - This class represents the playable levels within the game. These are instantiated from files created by the Level Editor or from hardcode addition of sprites and images to the level. The level acts as manager for the interaction and displaying of sprites (including the player) and always provides an exit point (IDoor) that acts as the goal of the level and terminates

the level.
    - In addition to the methods of IGameComponent, Level also implements
        - `public void addStartPoint (Location start)`
            - sets a mandatory start point for the level. This is the point at which the player will always start when entering the level.
        - `public Location getStartPoint ()`
        - `public void addSprite(Sprite s);`
            - adds the sprite s to the level so that it can interact with other sprites and be displayed.
        - `public void removeSpite(int x, int y);`
            - removes all sprites that intersect with the pass coordinate (x,y) from the level and from game play.
        - `public void setSize(Dimension size)`
            - Sets the size of the level. Most often, this will be bigger than the actual screen displaying the level and the scrolling manager will use the sze to determine when and where to scroll.
- IGameComponent.java
    - Defines methods of an element of the game. These are things such as Levels that the user players, Splash Screens, and possible (not yet implemented) things such as upgrade stores. These methods are
        - `public void update(double elapsedTime, Dimension bounds, GameView view);`
            - updates this game component based on the time and current bounds. For a splash screen, this might not do anything. For a playable level, this would update all of the sprites and other state inside of the level.
        - `public void addManager (LevelManager lm);`
            - adds a level manager to this GameComponent so that it can be exited and entered by the player.
        - `public void addInputListeners (Input myInput);`
            - adds input listeners unique to this game component to the current input. This is usually called when entering or starting a game component.
        - `public void removeInputListeners (Input myInput);`
            - removes input listeners of this game component from the current input. This is usually called when exiting a gamecomponent.
        - `public String getInputFilePath ();`

        - gives the location of the Input Resource file to use with the new input features unique to this game component.
    - `public void paint (Graphics2D pen);`
        - paints this game component. For a Splash page, this may be a single image. For a playable level, this will be painting all of the sprites, the background image, and any other features deemed necessary by the developer.
    - `public IDoor getDoor ();`
        - returns the exit IDoor for this game component. Every Game Component needs one since the IDoor specifies which level is the next level.
    - `public Player getPlayer ();`
        - Gives the current player in this game component. This is used for moving the player in between levels.
    - `public void addPlayer(Player p) ;`
        - sets the player in this game component to the player p. This is used for moving the player into this game component.
    - `public double get{Right, Left, Upper, Lower}Boundary ();`
        - returns the boundary of the game component. This is used to scroll around the game component. Most often for splash pages, this will just be the size of the view.
- Splash Page
    - This class is a useful class for game developer to either modify and use or use without changes. It currently provides a IGameComponent interface that displays a single image and has controls for
        - moving to the next level (currently pressing any key)
        - quitting the game (currently pressing 'q')

Level Management

- Level Manager
    - This controls the flow of levels in the game. This manages all of the levels, and maps levels in a linear fashion. This class only requires all of the levels to be used in the game, and it will handle all of the other functions of moving between levels.
        - Ex. Splash ->Level1 -> Level2->Splash
- Level Factory
    - Builds levels. This can either be done
        - hard coded, although this is difficult, it gives much more precision.

        - from files created by the Level Editor
    - Constructed levels are added to an instance of LevelManager.java
        - this usig LevelManager `lm.put(IDoor levelExit, level)`
            - here, the IDoor defines what level to proceed after the completion of the current level.
- IDoor.java
    - Defines methods to help the level manager direct levels. A class that implements IDoor is expected to act as a 'door' between two levels. The methods it requires to be implemented are:
        - `public void setNextLevel (IGameComponent level);`
            - Sets the level that this IDoor points to.
        - `public IGameComponent getNextLevel();`
            - gives the level that this IDoor points to.
        - `public void setManager(LevelManager lm);`
            - sets the LevelManager to help this IDoor move the player between levels.
        - `public void goToNextLevel(Player player);`
            - Takes the player to the next level.

Scrolling Management

- ScrollingManager
    - Determines how the level will scroll as the player traverses it. It controls the paint methods for all painted items including the background, player, and sprites.
        - `public void viewPaint();`
            - Does the calculations for the states that the background can be in and adjusts the painting for this as required.
        - `public Location playerPaintLocation();`
            - Since the location of the player is important in how all other sprites are painted, this method does the calculations where the Player should be painted then returns the Location. For example this is used when painting sprites because it is useful for translating the reference.
- OmniScrollingManager
    - This ScrollingManager provides a scrolling experience in any direction and when the edges of the Level are approached (as defined by Level.mySize), the scrolling will stop and allow the person to move naturally around that boundary.

- UniScrollingManager
    - This ScrollingManager provides a scrolling experience that will allow the player to traverse in any three directions but if they attempt to go “back” in the fourth direction they will be unable to. When this ScrollingManager is instantiated, it needs a Direction parameter (TOP, BOTTOM, LEFT, RIGHT) that indicates the directions in which the Player will be restricted from going (for example, in Mario the restrictive direction is to the Left).