Design Document

Software Engineering
Professor Do-Young Park

Team Members:
Emiz Intriago, Gurkiran Kaur, Wardah Aziz, Miranda Figueras
Team Leader:
Andrew Stephens

Table of Contents

1. Introduction

1.1 Purpose of the system

The purpose of our project was to create an enjoyable game that would be appropriate for all ages. We developed a 2D game that we named “Into the Woods,” which was designed with a similar gameplay style to “Mario Brothers” and some additional movement features reminiscent of “Ori and the Blind Forest.” We chose to give the player the ability to backtrack through the level, so that they could spend more time traversing a level searching for the key collectables. We planned to have multiple levels, where the player will search for and collect multiple keys, necessary to open the door, completing the level. There will be both neutral and hostile static obstacles which will prevent the player from quickly collecting the keys. After completing a level the game will automatically progress to the next level, except in the case that the highest level is completed leading to the main menu.

1.2 Design goals

Our game is written in Java using object-oriented programming concepts and it follows a “Model View Controller” architecture. We have attempted to write the software’s code which can be read easily, clearly showing the relationships between attributes, objects, and classes through the strategic use of abstraction and inheritance. The aforementioned has proven to be highly useful thus far and will allow us to commit changes or additions at a later time if needed. Through these appropriate programming practices, we expect to have a deliverable that supplies all of the proposed functionalities, that will be intuitive and perform as expected by the user.

Our goal for the client-side is to have created a game that is intuitive for the player. The user controls will follow the same structure as many other 2D platforming game which will make it simple for the player to learn the controls, while also providing the user with options for customization. In our Level design, we believe that the importance of the appearance of hostile deterrents in the level will make it obvious that they must be avoided. Alternatively, the appearance of collectibles should make the player immediately understand their significance and provide a beacon for the player to travel towards. Quality control has been extensive, and we expect that our game will execute without producing

either major bugs or crashes. In terms of graphical fidelity, we have utilized a delta time architecture in the game loop in order to provide both consistent update tick rate and render refresh rate, ensuring movement normalization across all game objects. This will provide a more pleasing game appearance and enhance the gameplay by preventing any stutters. We also expect that all visual feedback in the game will be what is necessary for the user such that there is nothing meaningless presented to them and irrelevant data will be hidden.

1.3 Comparison

1.3.1 Development Time vs Performance:

When developing this game, it really came down to the different platforms we had to use in order for our game to run and have trials where all the members of the group would have to pull the code to see if it would run without any issues with their PC's. To develop our game we used programs such as Intellij, GitHub, Git. The program is written in Java since we were most familiar with that language. This not only cuts development time because we don't have to deal with technical glitches that are directly linked to the impact of the performance of the game itself.

1.3.2 Space vs Speed:

Considering our game must be played instantaneously, we choose to use a large amount of space in order to ensure high speed. We retained the variables in numerous locations in memory to reduce time searching for them to keep the processor occupied computing game mechanics.

1.3.3 Functionality vs Usability:

The Player will control an avatar as they attempt to navigate among a forest of static platforms and hostile deterrents. We chose usability over functionality because we're making a game with simple yet fun interactions. Our game is quite easy to comprehend, and it also includes a how-to-play screen to help with usability.

1.3.4 Understandability vs Functionality:

Our system is expected to be simple to learn and understand. As a result, we had to reduce the game's complexity and functionality by removing confusing

and complex options so that players wouldn't waste time trying to figure out what they were doing and instead enjoy the simple, easy-to-understand game.

1.4 Criteria

1.4.1 Usability:

Into the Woods is expected to be a fun game for players, with a user-friendly experience that makes it simple for them to operate. For example, the game's basic and intuitive menu and interface will allow the player to concentrate on the game rather than the game's functions. The system will accept keyboard inputs from the users, making it simple for the gamers to utilize. It is evident that the majority of gamers almost always bypass the 'how to play' section of the game. However, because our system was designed in such a way that the Player will be able to understand it even without the aid of a how-to-play screen, it will be simple for them to grasp. From the player's perspective, it will be very simple to recognize and comprehend how the level ends, how the avatar advances, and the purpose of collectibles, among other things.

1.4.2 Performance:

One of the most essential design goals for games is optimization. We created a framework that manages concurrent threading, which helps to manage tick rates for both updating models and rendering views at a desired rate. This ensures that both object motion and calls to render canvases are consistent. This, in turn, reduces stuttering, balances game performance and restricts system resource allocation.

1.4.3 Extendibility:

The game's design allows us to add to and adjust its features and functionalities in the future based on user feedback or other factors. For example, we may be able to increase the game's difficulty by adding static entities, power-ups, and more levels.

1.4.4 Modifiability:

The source code of the system heavily uses abstractions and with abstractions come the ability to modify inheritance and function definitions at object instantiation. This provides very modifiable code which can be used across multiple avenues of the program.

1.4.5 Reusability:

Some of the subsystems in the game can be used in other games– namely the Gson and Javazoom libraries. However, the native subsystems are hardwired into our program, which in turn means that translating them onto another system would be quite a hassle. In order to make a component function in a different system, those components would need to be packaged in a separate library. One would import the library into their project to use it. In short, a number of our native subsystems can be used in other games since they are modular; but due to those subsystems being hardcoded for our system, it would be a dubious task.

1.4.6 Portability:

In general, portability is a significant feature of software because it expands the game's usability for consumers. To put it another way, it makes it possible to play the game on a variety of devices. As a result, we chose Java as the language because its JVM allows the system to be platform independent and ported to most laptops and desktops.

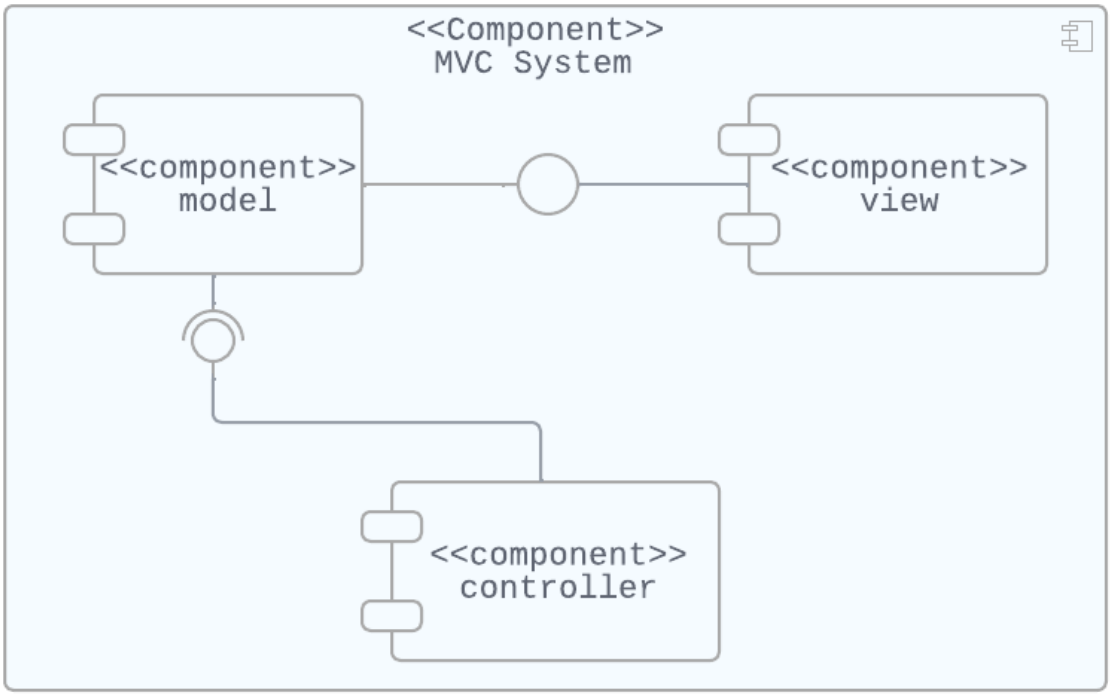
1.4.7 Performance Criteria:

It is critical for the game to quickly respond to the input requests made by the user. Our software was created in such a way it would respond very quickly to user input.

2. High-level software architecture

2.1 Subsystem Decomposition

We will be decomposing our system into three specific subsystems. This is with the expectation that we will later be able to modify the game more easily as well as to decrease coupling in our code, so that our code is not too interdependent. We have utilized the MVC Framework, or Model View Controller system design pattern. This means that we have seperated concerns into one of three buckets, either the Model, the View, or the Controller. The program logic is divided into these three elements which are loosely interconnected. The MVC Framework was a beneficial choice for our game design because inherent in it is that there is a separation of concerns because the code is separated based on function, categorized as either part of the controller, view, or model. This means that our code can be more loosely coupled.



The view is the subsystem which the user is able to see and to interact with. In our system, this bucket will include the menus as well as the game environment. These are UI and GUI representations of the game which are pleasing to the eye and can easily be understood by the user. The view subsystem contains the classes for both the main and pause menus as well as the game canvas and main window screen. The view will allow the user to select to start a new game, select a level to start the

game at, select the avatar they want to play with, change the visual and audio settings, quit the game, and provide help from the main menu. The user will also be able to view the pause menu at any time while in the game environment which will allow the user to view the help, change options, or return to the main menu. When the user clicks on one of the buttons on a menu, that selection is conveyed to the controller.

The controller is the second subsystem and contains the code that allows our game to respond to the user's inputs, which were captured by the view. The controller also is the interface working between the view and the model. The model is the component of the subsystem that contains all the game related logic necessary to run gameplay. In "Into the Woods'" The controller is responsible for interacting with the user. When a user clicks on a button, or uses a key input, the controller decides what to do in response to that input. The controller contains the information needed to react to user input, through both button clicks using a mouse as well as key inputs. This allows the user to move through the menu options and to control the avatar in the game environment.

The model subsystem is the third bucket, and will implement the domain logic. The model subsystem contains the core data for our game, the data model, and the ability to load the game, save a game, and load a saved game. The logic stored in the model is independent of the view, or user interface.

2.2 Hardware/software mapping

The game will run on any operating system that can run Java such as Windows, Mac or Linux. The game only requires the basic peripherals for a computer; Which are a pc, a display, a keyboard and a mouse. Audio output is not required, but it will enhance the experience.

The programming will be done under the JAVA JDK version 18 and it will be done using an IDE called Intellij. Therefore the user will need to download either JDK Version 18 or Java 8 in order for the game to run properly. There are two important additions in the gaming program. One of the most important libraries is Gson. Gson is a simple Java-based library which works to serialize Java to JSON and vice versa. Gson will be used to import any sprite sheet that will be implemented in the game. Another external library would be a java.zoom which allows the game to have various audio. Depending on if the user has clicked on a button of background music.

2.3 Persistent data management

The game will create game instances, like objects, spikes etc. at runtime and store them into memory. Such instances cannot be modified. However, some data like user preferences, sound settings, and latest score will be kept in a modifiable text file, therefore users can change or update the values of these files during game time. Graphics and music files will be located within the classpath and used in various parts of the game to enhance the gameplay experience.

2.4 Access Control and Security

No internet/network connection or database will be used. Anyone can play the game once the Jar executable has been downloaded. As a result, no access control or security measures will be in place to prevent data leaks. Due to .jar files behaving like .zip files do, all source code and attached files can be naturally decompressed and extracted for observation. Obfuscation methods have not been implemented to skirt this behavior.

2.5 Boundary Conditions

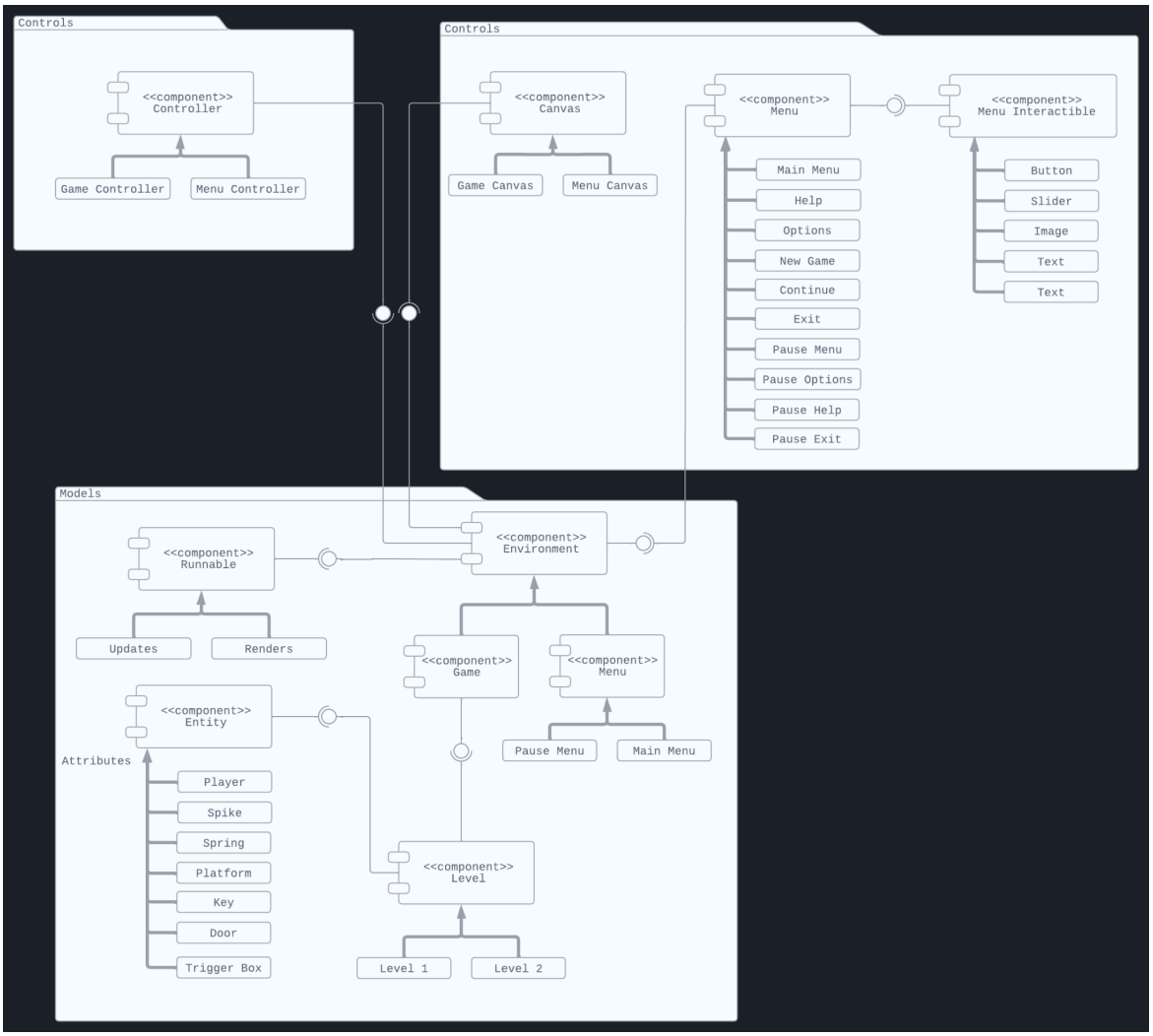
2.5.1 Termination

When a user decides to exit from the game while in gameplay, user can click their “Escape'' button on the keyboard to return to a menu from there a person can either continue to resume the or click'' Back to main menu”. From there the game can be terminated by clicking the “Quit to Desktop” button in the main menu.

2.5.2 Failure of the Game

If an issue occurs during the file read, the game may begin without sound or visuals. The only option that can resolve this issue is for the user to redownload the JAR file . If the game crashes during play due to a performance or design fault, the data (current score) may possibly be lost.

3. Subsystem Services



3.1 View Subsystem

This Subsystem is in charge of controlling and administering the game locally rather than externally. The *View* subsystem provides components that listen for user instruction, pass data to the *Model*, and display user feedback. The *View* subsystem incorporates a number of derived classes:

1. *AMenu*
2. *ACanvas*
3. *AWindows*
4. *AMenuComponent*

When the game is started, the player is met with the very first type of *View*, the *AWindow*. This *AWindow* is displayed and acts as the container for all nested *View* components. Within the *AWindow* resides an *ACanvas*, which acts to display graphics and control the buffer strategy. Within this *ACanvas* is a menu-subtype AEnvironment called an *AMenuEnvironment*. An *AMenuEnvironment* contains *AMenu* views. *AMenu* objects are displayable containers for intractable views called *AMenuComponents*. *AMenuComponents* are subtyped, and are composed of individual functionality to control either the navigation between pages of an AMenuEnvironment or extra functions for that page.

In the Main Menu, the user may select from a variety of choices such as New Game, Continue Game, Options, Help, Quit, and so on. Multiple types of *AMenu's* exist in the AMenuEnvironments, each of which are separate pages with its own set of buttons and subpages. As a result, separate menus might lead to other menus or plainly execute specific functions. *AMenu's* are all housed within what is known as the *AMenuEnvironment*. There is a stack of distinct menu pages which contains abstract menus it keeps and orders the array of menus we are currently using. Essentially, *MainMenuEnvironment* includes a stack of *AMenu's* and allows you to manage the flow of *MenuModels* that you are now utilizing. *AMenuModel* is utilized in both *MainMenuEvironment* and *PauseMenuEnvironment*, and both of these items require *AMenuEnvironment,* which facilitates menu navigation. *AMenuEnvironment* and *GameEnvironment* are both examples of environments (or *AEnvironments)*. They may all be utilized independently of one another. Controllers are dictated by *PauseMenuEnvironment* and *MainMenuEnvironment. AMenuImage* and *AMenuBotton* are menu components that exist inside of *AMenu* and they are updateable which is all based on the classes inside the *AMenu*.

ACanvas inherits the JPanel class which is something that the program can draw to just like a canvas in real life. This is an abstract class that inherits *JPanel* which contains more methods inside of it. It acts as a wrapper class for all other abstract classes. *MenuCanvas* and *GameCanvas* are two separate canvases that derive from *ACanvas* which means that they contain the *JPanel* characteristics. They contain references to *ACanvas* and they have graphics drawable with them. They exist separately because we want to draw *MenuCanvas* and *GameCanvas* separately and asynchronously against their updates. The *MenuCanvas* and *GameCanvas* are both redrawn every certain number of ticks. This happens when you update the game objects you must redraw the frames.

Finally, *AWindows* is an abstract class to rename a JFrame and it also contains some methods that the *MainWindow* will use.

3.2 Control Subsystem

AControlsModel is an abstract class that contains *AKeyController* and *MouseController*. This structuring of controls will adapt to either allowing the player to modify the avatar's direction in the game, such as up, down, left, and right, or to interact with buttons in the menu screens.

GameControls has given the user the ability to make their avatar leap and run for navigation throughout the game.

MenuControls gives the user the ability to navigate throughout all of the *AMenuEnvironment* components.

AKeyController is an abstract class which will listen for Key Input based on KeyCode. A matching key code will set the status of that key press to true. When a key is released, the directional is set to false in the key released code. The state of the Keys are recorded based on user input.

AMouseController is an abstract class that implements the *MouseListener* and *MouseMotionListener* interfaces. The code for this class is fairly comparable to that for *AKeyController* as it resets the "mouse" click and release. The input is followed by Actions, which accepts direct input and converts it to programming logic. By pushing the left mouse's side or the right mouse's side, the user can control the recorded position of the mouse which is accessed by components which listen for that position.

MenuMouseControls is a subclass of *AMouseController* that allows the mouse to maneuver around the AMenuEnvironments and interact with the *AMenuComponents* contained within them.

GameMouseControl allows the mouse to handle many of the game's actions, with the exception of character movement. Future plans involve interactable entities in the level which act on user clicks.

MenuKeyControls extends from *AKeyController*. Key pressed (true) and key released (false)

GameKeyControls is a subclass of *AKeyController*. There are two parameters in *GameKeyControls*. The first is termed *keyPressed* and is true, while the second is called *keyReleased* and is false. This essentially defines the keys that the user can use to move the avatar.

3.3 Model Subsystem

The model subsystem contains the models necessary for our game to run and for game information to be saved. The model subsystem contains the abstract classes for *AEnvironment* and *APhysics* which are needed to run the game environment and menu environment. *APhysics* contains the information for the function of the platforms and triggers in each level, and what the response should be when the avatar collides with one of those things. The platforms are the surfaces where the avatar is able to walk or jump off of, while the triggers represent the collectible keys, the hostile deterrents, and the springs. *ALevel* contains the information for where each of these will be in the game levels.

The abstract class *AMenuEnvironment* is a subtype of the *AEnvironment* and contains the controls for the menu environments and the associated contained entities. The *MainMenuEnvironment* and the *PauseMenuEnvironment* are subtypes of the *AMenuEnvironment* class.

AOverlayComponent is a public abstract class that is extended by the *MapOverlay* and *StatsOverlay* private classes. The *StatsOverly* holds the information shown in the HUD, or heads up display, recording the keys collected by the player. The *MapOverlay* contains the information for the entire level map and the current location of the avatar, which the player will be able to see through the view. The *PlayerInventory* will record the number of collectible keys that the player has picked up.

APropTrigger was used to standardize the functionality of the objects with which the character interacts through collisions. These triggers can be caused by colliding with the spikes, springs, or the key collectibles. *Aprop* is an abstract class that functions as a wrapper class for the different level props (springs, spikes, and key collectibles). The *Atriger* and *AReactProp* are abstract classes which describe what happens when the character actually collides with the different props. When the character collides with the spikes, the character dies and the level is restarted, the springs can be used to jump further onto out of reach platforms, and the key

collectibles are captured when collided with, being stored in the HUD until three are collected and the level can be completed.

AActor extends *APhysics* and is the most basic game entity in the game environment. The *AActor* abstract class is a superclass with *APawn* and *Aprop* as subclasses. *APawn* is an abstract class which represents a game object that has the ability to move, for example the game character which is controlled by the user. The *ACharacter* abstract class is a subclass of *APawn* and it allows the user to control the character avatar.

The *AFileReader* abstract class is necessary for opening a saved game that the user was playing. After one or more levels have been completed, the game can be saved so that the levels completed will be recorded and that when the user re-opens the game, they can continue playing with the next available level. The *AFileReader* is able to accept and store a file where the read method is defined at instantiation.

The abstract class *ARunnable* is a superclass that implements the subclasses *RenderRunnable* and *UpdateRunnables*, and it implements the *Runnable* class. The *RenderRunnable* class contains the information for the loop that dictates the render rate and the *UpdateRunnable* contains the continuous loop that dictates the update rate. These are important parts of all of the environments.

4. Low-Level Design

4.1 Object design trade-offs

The tradeoffs of using design patterns exist as issues if the design was not implemented intelligently. In the interest of creating a complex system, necessary thought was put into the choice of such design patterns.

4.1.1 Abstract Factory Design Pattern

There are a number of areas within the system which heavily utilizes the robust library of abstract classes. These classes allow for less strict means of defining functions, and allow for dynamically defining said definitions at the time of object creation. This helped with making numerous child classes, but created a readability issue where behavior is relative to the location of creation.

Subsystems which notably use an Abstract Factory Pattern to make use of such abstract classes include the *AMenu* and the *ALevel* classes. These container classes rely on the use of contained *AMenuComponent* objects and *AActor* abstract classes respectively, defining such aggregates based on the role of that particular container class.

For example, the *AMenu* defines a list of *AMenuComponents*, where each *AMenuComponent* added to the list might be of different types, such as the *AButtonView* and *ASliderView*. While the *AButtonView* contains specific functions, one function may be overridden at creation to deal with a particular use case. We can see this in the *onClick(int, int)* method within the *AButtonView*, where one instance of the method might send the user back one menu page while another instance of the method might terminate the program entirely.

4.1.2 Singleton Design Pattern

With the need to minimize the possibility of null pointer errors, the decision to use a Singleton Pattern came into play. Though this made management much easier, it also impacted the load times at startup and memory management is low. However, load times after initialization are quite fast.

This singleton pattern is heavily utilized in the Main class, whereby numerous object instances are first created. Then, the instances are initialized with other specific related instances passed into them.

Not only does the singleton pattern assist with preventing null errors, it also assists with tracing. This helped to locate problematic code which usually came in the form of logic errors.

4.1.3 Facade Design Pattern

The system incorporates numerous subsystems which handle large chunks of code, so the Facade Design Pattern was practically required. This design pattern is very important for delegation of duty, making sure that the control is centrally located. However, this required passing references to the containing class upon object initialization which proved to cause tracing issues due to pathing between numerous methods between encapsulated objects to get to the top layer object.

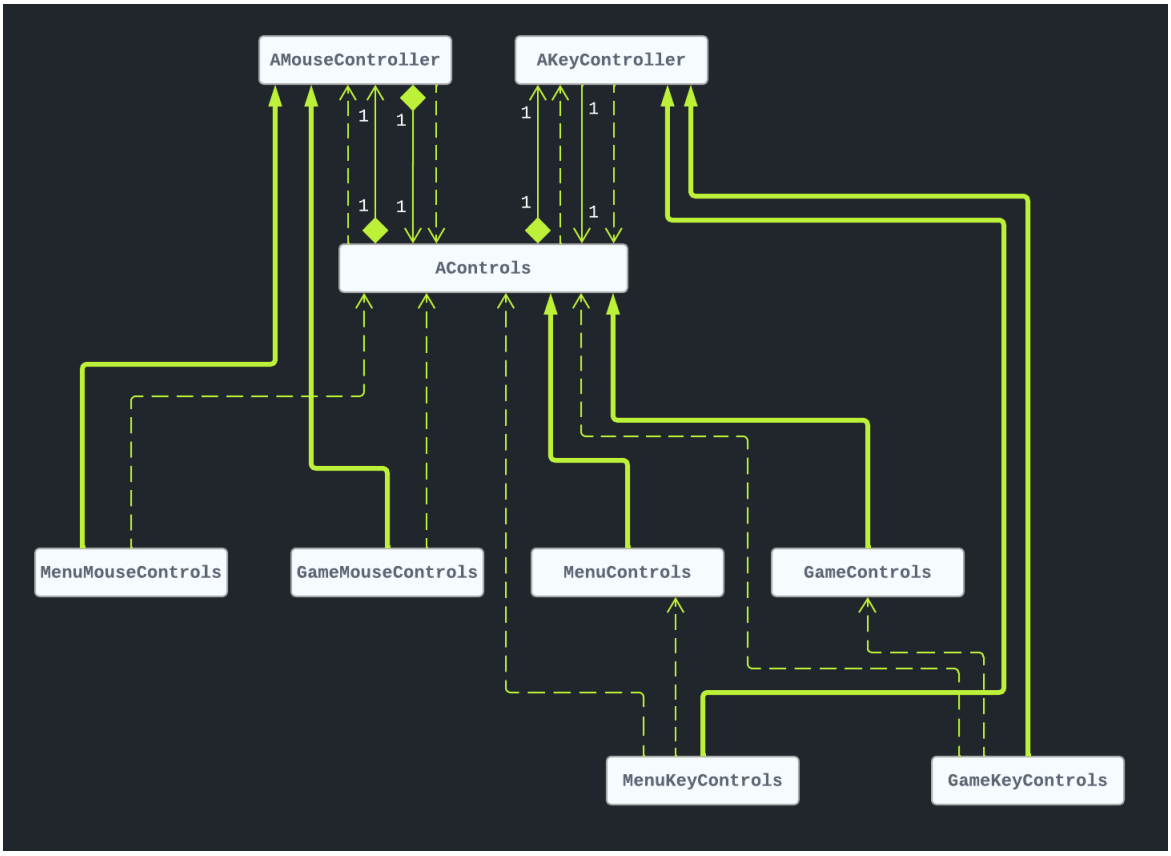
The heaviest use of this design pattern resides within the *EnvironmentsHandler* class. The class acts like the root between all existing AEnvironment instances, whereby navigation between environments is called and executed. It contains the logic required for controlling the threads involved with particular environments as well as controls the communication between the display window should the user request configuration changes.

4.1.4 Prototype Design Pattern

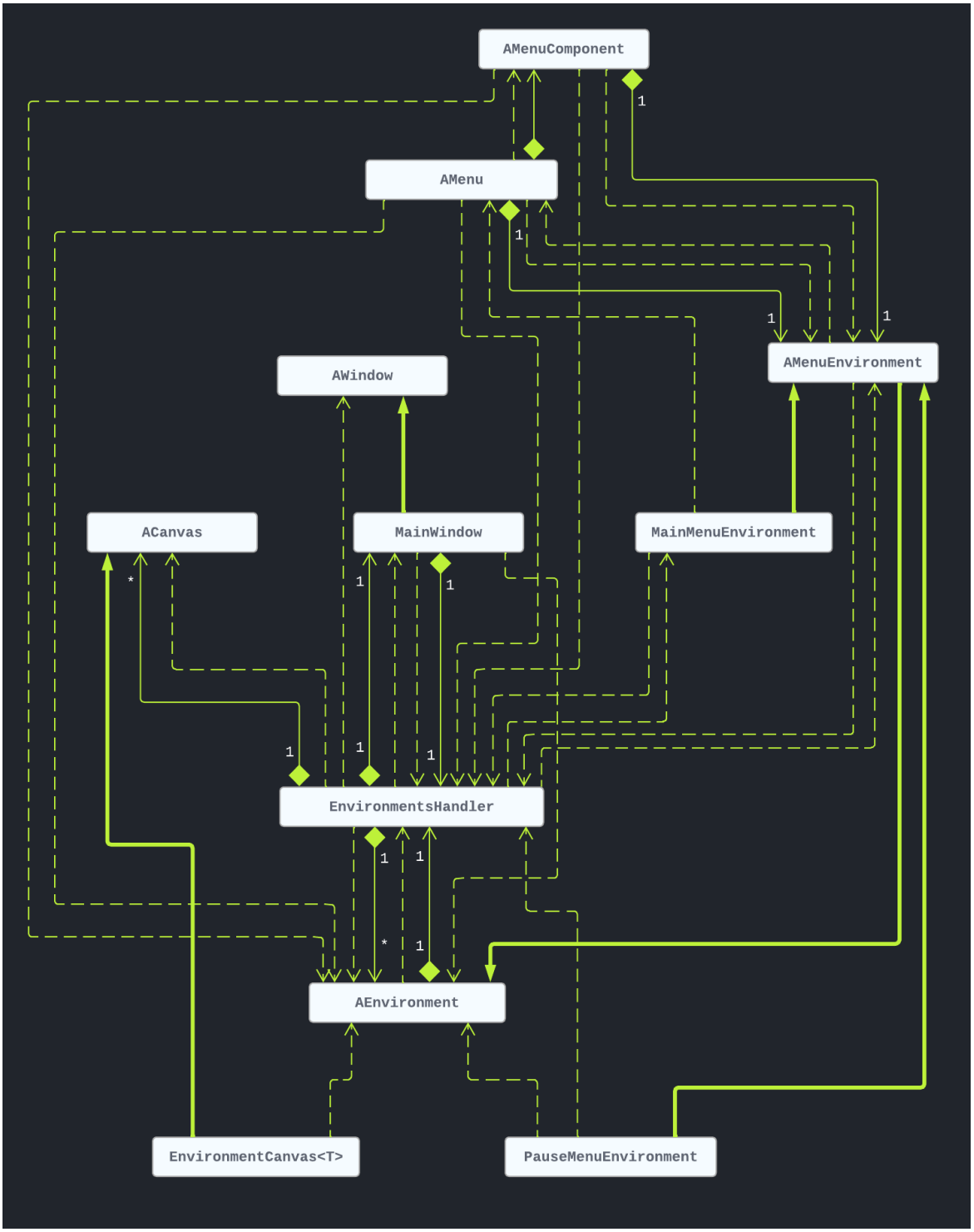
The prototype design pattern is used for nearly all models, controls, and views which derive their structure from a parent class. This is largely notable in classes which stem from the *APhysics* class which contains the logic for boundary collisions. There are a large number of such deriving classes and they all behave differently. While helpful, the prototype design pattern proved hard to maintain as all nested abstract classes require checking superclasses for their operations to see what problems were being caused for inheriting classes.

4.2 Final Object Design

4.2.1 Controls

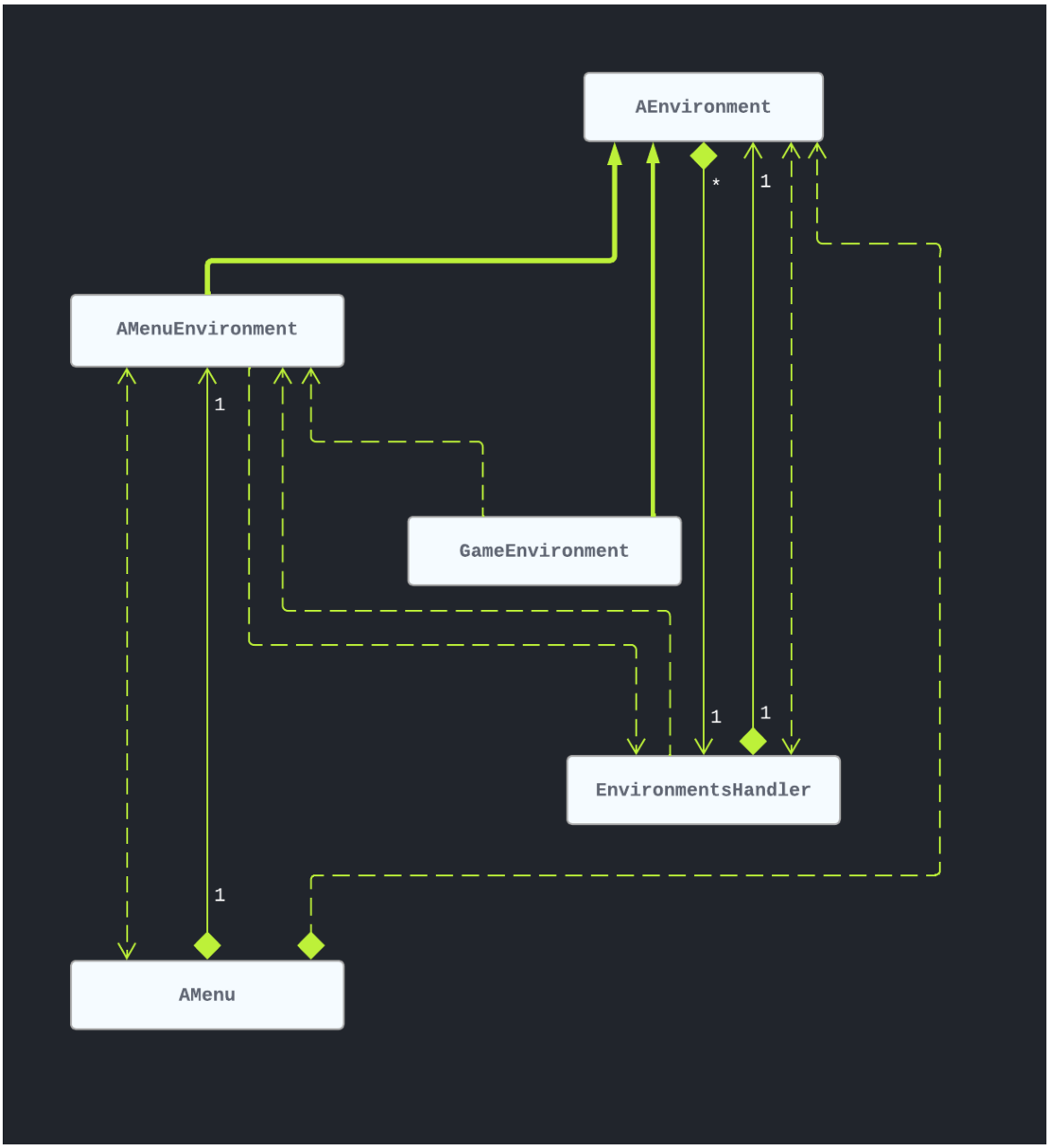


4.2.2 Views

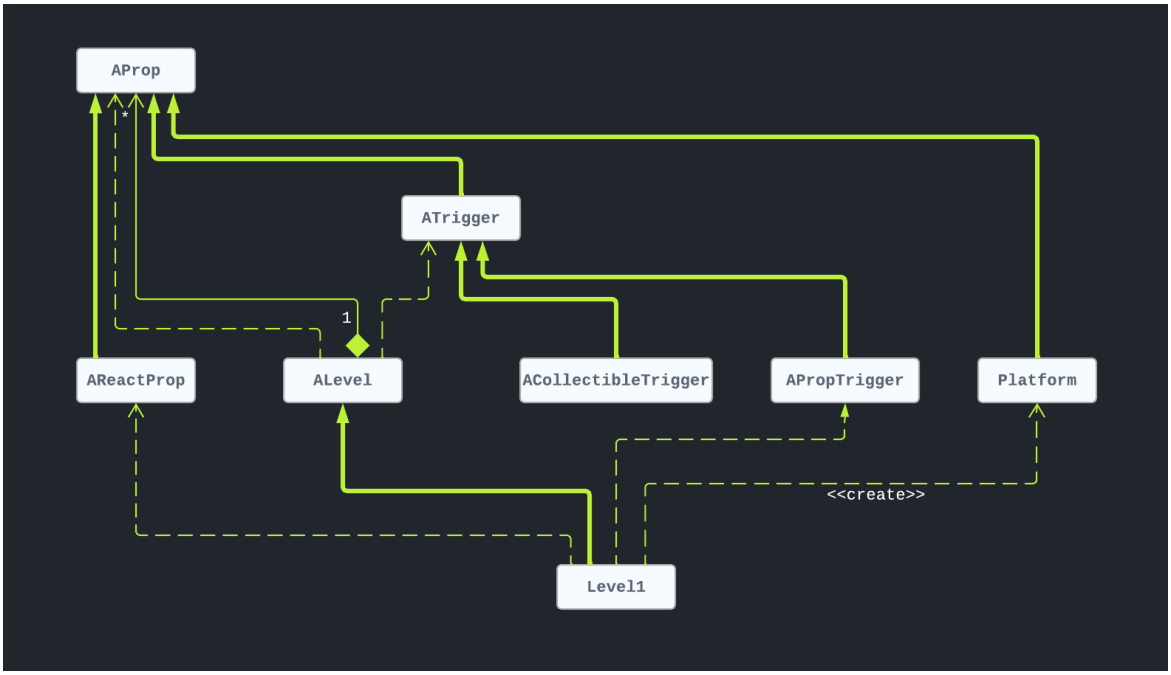

AMenuComponent
AMenu
AMenuEnvironment
AWindow
ACanvas
MainWindow
MainMenuEnvironment
EnvironmentsHandler
AEnvironment
EnvironmentCanvas<T>
PauseMenuEnvironment

4.2.3 Models

4.2.3.1 Environments



4.2.3.2 Game Entities



4.3 Packages

4.3.1 Internal Packages

4.3.1 controls

This package contains classes for mouse and keyboard input controls.

4.3.2 main

This package contains the Main class.

4.3.3 models

This package contains subpackages dealing with the Models subsystem.

4.3.3.1 actors

This package contains classes for all top-level entities.

4.3.3.2 camera

This package contains the Camera class.

4.3.3.3 environments

This package contains classes which deal with all defined environments.

4.3.3.5 levels

This package contains the Levels of the game.

4.3.3.6 prototypes

This package contains all abstractions for top-level entities.

4.3.3.6.1 actor.pawn.character

This package contains the abstract class of the PlayerAvatar.

4.3.3.6.2 components

This package contains the abstract classes for menu and game interactables.

4.3.3.6.3 controls

This package contains the controls abstract classes.

4.3.3.6.4 environments.menu

This package contains abstractions of all environments and menus.

4.3.3.6.5 level.prop

This package contains abstractions of all level props.

4.3.3.6.6 threading

This package deals with the abstract runnable class.

4.3.3.6.7 views

This package contains abstracts for all top-level swing-derived classes.

4.3.3.7 runnables

This package contains the classes of render and update runnables.

4.3.3.8 sprites

This package contains classes which handle sprite creation.

4.3.3.9 utils

This package contains subpackages of extra utilities.

4.3.3.9.1 audio

This package contains utilities for audio control.

4.3.3.9.2 config

This package contains the Config and other data classes.

4.3.3.9.3 drawables

This package contains the drawable interfaces.

4.3.3.9.4 files

This package contains the classes which deal with parsing files.

4.3.3.9.5 physics

This package contains the Physics class.

4.3.3.9.6 resources

This package contains the Resources class.

4.3.3.9.7 updates

This package contains the update interface.

4.3.4 views

This package contains the classes which derive from the abstract view classes.

4.3.2 External Packages

4.3.2.1 com.google.gson

This package is included through Gson's libraries. It contains utilities for parsing through Json files.

4.3.2.2 java

These packages are included in the proprietary Java library.

4.3.2.2.2 awt

This package deals with Java graphics.

4.3.2.2.2 util

This package contains utilities for algorithms and data structures.

4.3.2.2.3 io

This package deals with IO implementations, such as File reading.

4.3.2.2.4 net

This package assists with protocols for networks.

4.3.2.3 javax

The extension package for the Java libraries.

4.3.2.4 javazoom.jl

The package used for a simple audio player for MP3 files.

4.4 Class Interfaces

Subsystem: Controls

AControls

public abstract class **AControls**:

AControls is the parent class of all Environment-specific Controls. Contains the state of Actions and state of movement controls as well as references to the Mouse and Key Controls which make up the Environment's controls.

Fields:

protected AKeyController keyController:
The KeyControls related to this AControls parent.

protected AMouseController mouseController:
The MouseControls related to this AControls parent.

protected boolean[] directionals:
The directionals states.

protected boolean[] actions:
The actions states.

Methods:

public void init(*AMouseController*, *AKeyController*):
Initializes the AControls model. Assigns AMouseController and AKeyController subtypes.

public AKeyController getKeyController():
The KeyController that's specific to a subtype of AEnvironment which this object is built for.

public AMouseController getMouseController():
The MouseController that's specific to a subtype of AEnvironment which this object is built for.

public void setDirectional(*AControls.Directionals*, *boolean*):
Accepts the Directional type and the state that the Directional is to be set to.

public boolean[] getDirectionals():
The state of all Directionals.

public void setAction(*AControls.Actions*, *boolean*):

Accepts the Action type and the state that the Action is to be set to.

public boolean[] getActions():
The state of all Actions.

public boolean getAction(*AControls.Actions*):
Obtains the state of the accepted Action type.

public void resetAction(*AControls.Actions*):
Resets the specific Action to a disabled state.

public void reset():
Sets the state of both Directionals and Actions to disabled.

GameControls

public class **GameControls**
extends ***AControls***:

GameControls is dedicated to the Controls of the GameEnvironment

Fields:
boolean[] abilities:
The enumeration of all Abilities.

Methods:
public void setAbility(*GameControls.Abilities*, *boolean*):
Controls the state of a specific ability.

public boolean[] getAbilities():
The current state of all Abilities.

public void reset():
Sets states of all controls to disabled.

MenuControls

public class **MenuControls**
extends ***AControls***:

MenuControls is dedicated to the Controls of the AMenuEnvironments, such as MainMenuEnvironment and the PauseMenuEnvironment.

Methods:
public boolean isAction(*AControls.Actions*):
Accepts an action and checks if the current controlled state of the action is set to enabled.

AMouseController

public abstract class **AMouseController**

implements ***MouseListener***, ***MouseMotionListener***:

The AMouseController is an abstract class that Mouse controller based class inherits.

Fields:
protected AControls controlsViewModel:
The parent AControls Model

protected final int[] mPos:
The recorded position of the mouse.

protected boolean isLeftPressed:
If the left mouse button is pressed.

protected boolean isRightPressed:
If the right mouse button is pressed.

Constructor:
protected AMouseController(*AControls*):
Creates the AMouseController.

Methods:
public boolean isLeftPressed():
Checks if the left mouse button is pressed.

public boolean isRightPressed():
Checks if the right mouse button is pressed.

public void setPos(*int*, *int*):
Records the x and y position of the mouse pointer.

public int[] getPos():
Gets the saved position of the mouse.

public void reset():
Resets the left and right button.

GameMouseControls

public class **GameMouseControls**
extends ***AMouseController***:

Dedicated GameEnvironment Controls class that extends the AMouseController.

Constructors:
public GameMouseControls(*AControls*):
Creates the GameMouseControls and references the parent AControl.

MenuMouseControls

public class **MenuMouseControls**
extends ***AMouseController***:

Dedicated AMenuEnvironment Controls class that extends the AMouseController.

Constructor:
MenuMouseControls(*AControls*):
Creates the MenuMouseControls and references the parent AControls.

AKeyController

public abstract class **AKeyController**
implements ***KeyListener***:

The AKeyController is an abstract class that the Key controller based class inherits.

Fields:
protected AControls controlsModel:
The parent AControls model.

Constructor:
public AKeyController(*AControls*):
Creates a new AKeyController.

Methods:
public AControls getControlsModel():
Gets the parent ControlsModel.

MenuKeyControls

public class **MenuKeyControls**
extends ***AKeyController***:

Dedicated AMenuEnvironment Controls class that extends the AKeyController.

Methods:
public boolean isAction(*AControls.Actions*):
Checks if the action passed is an action.

public void reset():
Resets the control Model

GameKeyControls

public class **GameKeyControls**
extends ***AKeyController***:

Dedicated GameEnvironment Controls class that extends the AKeyController.

Constructors:
public GameKeyControls(*AControls*):
Creates the GameKeyControls and references the parent AControls.

Subsystem: Views

AMenu

public abstract class **AMenu**
implements ***IUpdatable***, ***IDrawable***:

The AMenu is a superclass to all menu pages. This is a dynamic class which enables unlimited AMenuComponent additions. This class exists to deliver a seamless experience to the user with little cost to performance. The AMenu handles user input and either passes the input to each AMenuComponent. Or handles the request locally, depending on the input made. All AMenus exist within an AMenuEnvironment. To be seen or controlled, they must exist at the top of the environment's stack.

Fields:
protected float centerH:
The centered position of the screen.

protected float centerW:
The centered position of the screen.

private ArrayList<AMenuComponent> components:
The Menu Buttons.

protected BufferedImage image_buttonRect:
Background Image.

private AMenuEnvironment parentEnvironment:
The Parent Menu Model.

Constructors:
private AMenu(*AMenuEnvironment*):
Assigns the Menu with the parent AMenuEnvironment for reflection. Should be called through subclass inheritance.

Methods:
public boolean registerInput():
Obtains the controller and listens for specific input and executes the content if input is accepted.

public AMenuEnvironment getParentEnvironment():
Gets the AMenuEnvironment containing this Menu

public EnvironmentsHandler getEnvironmentsHandler():
Gets the EnvironmentsHander that contains this AMenuEnvironment.

public Resources getResources():
Gets the Resources.

public AMouseController getMouseController():
Gets the MouseController for the containing Environment.

public AKeyController getKeyController():
Gets the KeyController for the containing Environment.

public void addComponent(*AMenuComponent*):
Accepts an AMenuComponent object and adds it to the components list.

public void reset():
Resets all the contained AMenuComponents.

HelpPage

public class **HelpPage**
extends ***AMenu***:

The Help Page, containing text content to assist the player.

Constructors:
public class HelpPage(*AMenuEnvironment*):
Builds the Help page.

LevelSelectPage

public class **LevelSelectPage**
extends ***AMenu***:

Level Select Page. Allows the user to choose a previously completed level, or a level that they need to complete next.

Constructors:
public class LevelSelectPage(*AMenuEnvironment*):
Builds the Level Select page.

MainMenuPage

public class **MainMenuPage**
extends ***AMenu***:

Level Select Page. Allows the user to choose a previously completed level, or a level that they need to complete next.

Constructors:
public class MainMenuPage(*AMenuEnvironment*):
Builds the Level Select page.

Methods:
public void navigateToLevelSelectPage():
Navigates to the specified level and swaps the Environment.

NewGamePage

public class **NewGamePage**
extends ***AMenu***:

The New Game Page. Contains the character selection and the option to begin a fresh play-through.

Constructors:
public class NewGamePage(*AMenuEnvironment*):
Builds the New Game page.

OptionsPage

public class **OptionsPage**
extends ***AMenu***:

Gives the user the option to set their target frame limit, their Window style, and the size of their window.

Fields:
Private final AButtonView button_apply:
The apply button used to accept changes.

private short selectedFramerate:
The user-defined framerate

private Config.WindowType selectedWindowType:
The user-defined window type

private int selectedWindowWidth:
The user-defined window dimension

private int selectedWindowHeight
The user-defined window dimension

Constructors:
public class OptionsPage(*AMenuEnvironment*):
Builds the Options page.

QuitPage

public class **QuitPage**
extends ***AMenu***:

A Confirmation to ensure the player's intention.

Constructors:
public class QuitPage(*AMenuEnvironment*):
Builds the Quit page.

StartScreenPage

public class **StartScreenPage**
extends ***AMenu***:

This is the default landing page on startup. Contains a single menu button that says "start".

Constructors:
public StartScreenPage(*AMenuEnvironment*):
Creates the Start Screen Page

Methods:

public MainMenuPage navigateToMainMenuPage():
Navigates the user to the Main Menu Page

PauseExitPage

public class **PauseExitPage**
extends ***AMenu***:

A Confirmation to ensure the player's intention.

Constructors:
public class PauseExitPage(*AMenuEnvironment*):
Builds the Exit page.

PauseHelpPage

public class **PauseHelpPage**
extends ***AMenu***:

The Help Page, containing text content to assist the player.

Constructors:
public class PauseHelpPage(*AMenuEnvironment*):
Builds the Help page.

PauseMenuPage

public class **PauseMenuPage**
extends ***AMenu***:

The landing page of the Pause Menu. Contains menu button components for navigation into sub pages.

Constructors:
public class PauseMenuPage(*AMenuEnvironment*):
Builds the paused main page.

ACanvas

public abstract class **ACanvas**
extends ***JPanel***:

The abstract wrapper class for an environment's canvas.

Methods:
public abstract void render():
The abstract wrapper class for an environment's canvas.

EnvironmentCanvas

public class **EnvironmentCanvas** <T extends AEnvironment>
extends ***ACanvas***:

The EnvironmentCanvas behaves ubiquitously between all types of Environments.

Fields:
private T environment:

The generic type for the Environment that will be rendered.

Methods:
public void init(*T*):
Initializes this Canvas with the Environment type that is specified in the generic.

public void render():
The render method that all ACanvases must define.

AWindow

public abstract class **AWindow**
extends ***JFrame***:

AWindow is the parent class of MainWindow and exists to separate external functionality from the MainWindow. This class extends JFrame, which means that the AWindow inherits the functionality from that class. This class holds the business logic for constructing the window and building the cursor.

Fields:
private Resources resources:
The Resources that stay persistent across all objects.

Methods:
public void setResources(*Resources*):
Sets the Resources reference for the AWindow

public void buildCursor(*boolean*):
Constructs the Window based on user preferences. Uses the final design to set the actual dimensions to the Config data.

protected void constructWindowAndDimensions(*Config*):
If the cursor is visible, we build a new cursor to replace the standard system cursor with an image saved within the resources director.

MainWindow

public class **MainWindow**
extends ***AWindow***:

MainWindow class inherits from the AWindow class. MainWindow contains references to the EnvironmentsHandler. This class is used to build a new Window as a JFrame, and allows the structuring of such a window to the specifications set by the user. It allows for the creation of a custom cursor as well as rebuild the window based on user preferences.

Fields:
private EnvironmentsHandler environmentsHandler:
The EnvironmentsHandler connected to the window.

Methods:

public void init(*Config*, *EnvironmentsHandler*):
Initializes the Main Menu with a reference to the EnvironmentHandler. Also constructs the Window and builds the cursor for the window.

public void clearComponents():
Removes all components from the window before rebuilding.

public void build():
Adds the necessary components to the window.

AMenuComponent

public abstract class **AMenuComponent**
implements ***IUpdatable***, ***IDrawable***:

AMenuComponent is the parent class of all Menu components. They are both updated and drawn. In their basic form, they contain positional data and size data relative to the default screen dimensions. Text may be present in the component if allowed. Sound may also emit if a trigger is present. Contain a spriteSheet, foreground and backgrounds, background images, and tints. Nested Class Summary

Fields:
protected final AMenuComponent.ImageScale scaleType:
The Parent menu model which holds all pages and subpages.

protected BufferedImage backgroundImage:
The background image of the component.

protected BufferedImage tint:
The image's tint of an image in the component.

protected SpriteSheet spritesheet:
The SpriteSheet that will be used to draw this component.

protected Color foregroundColor:
The foreground color overlaying the other parts of the component.

protected Color backgroundColor :
The background color under the other parts of the component.

protected String text:
Text to be displayed. Will be removed if we add actual images in place of awt views.graphics.

protected int x:
The horizontal position of the component.

protected int y:
The vertical position of the component.

protected int w:
The width of the component.

protected int h:
The height of the component

protected boolean isFocused:
If the component has focus from the user.

protected boolean isPressed:
If the component has been activated by the user.

protected boolean playSound:
If the component should play audio.

Constructors:
public AMenuComponent(*AMenuModel*):
Initializes the AMenuComponent

Methods:
protected AMenuEnvironment getParentEnvironment():
Gets the parent AMenuEnvironment that contains the Menu of this component.

protected EnvironmentsHandler getEnvironmentsHandler():
Gets the parent EnvironmentsHandler that contains the Menu of this component.

protected boolean isInBounds(*float*, *float*):
Checks if the coordinates passed are within the bounds of this component. Typically used with the mouse coordinates. Typically called from the registerInput() methods.

protected void setIsFocused(*boolean*):
Sets if the component is being hovered over.

public abstract boolean onClick(*float*, *float*):
Defined by the subtype component or by abstract creation. Meant to be used on a click event.

public void draw(*Graphics*):
Allows for standard draw calls from a Canvas object.

public void updateSpriteSheet(*float*):
Updates the spriteSheet to the next frame.

public int right()
Gets the right side of the component

public abstract void registerInput():
Defined by the subtype component or by abstract creation. Meant to be used on an update method to check for any user input from the current controller.

public void setBackgroundImage(*BufferedImage*):
Sets the background image

public void setImageScaling(*AMenuComponent.ImageScale*):
Sets the Image Scale

public void setTint(*String*):
Sets the text of the component.

public void setSpritesheet(*SpriteSheet*):
Sets the spriteSheet for the component animation.

public void setPlaySound(*boolean*):
Sets if the component can play audio.

public boolean isPressed():
Gets if the component is pressed.

public int getX():
Gets the X coordinate.

public int getY():
Gets the Y coordinate.

public int getW():
Gets the Width of the component

public int getH():
Gets the Height of the component.

public void reset():
Resets the spriteSheet and the state of the component to default.

public void update(*float*):
Update method accepts the actual/target ratio to regulate the update rate.

AImageView

public class **AImageView**
extends ***AMenuComponent***:

The AImageView is a class that simply displays an image based on the rendering procedure specified.

Constructors:
public AImageView(*AMenuModel*, *int*, *int*, *int*, *int*, *BufferedImage*, *AMenuComponent.ImageScale*):
Initializes the AImageView

Methods:
public boolean onClick(*float*, *float*):
Defined by the subtype component or by abstract creation. Meant to be used on a click event.

public void draw(*Graphics*):
Allows for standard draw calls from a Canvas object.

public void update(*float*):
Update method accepts the actual/target ratio to regulate the update rate.

public void registerInput():
Defined by the subtype component or by abstract creation. Meant to be used on an update method to check for any user input from the current controller.

AButtonView

public abstract class **AButtonView**
extends ***AMenuComponent***
implements ***IDrawable***, ***IUpdatable***:

The AButtonView is a component that allows for user interaction in the form of mouse clicks. The button is constantly checked for an activated state. If it is active, it will execute a procedure defined in its definition.

Fields:
public boolean isEnabled :
If the button is currently enabled, or used.

Constructors:
public AMenuButton(*AMenuModel*, *int*, *int*, *int*, *int*):
Initializes the AButtonView

Methods:
public void setX(*int*):
Sets the x position

public void playSound():
Plays the button click audio sound.

public void registerInput():
Checks for the mouse input and does actions based on positive recognition.

public void setEnabled(*boolean*):
Checks if the button is enabled.

public void reset():
Resets the button's audio state

public void draw(*Graphics2D*):
Allows for standard draw calls from a Canvas object.

ATextView

public abstract class **ATextView**
extends ***AMenuComponent***
implements ***IDrawable***, ***IUpdatable***:

The ATextView component simply should draw text within the component.

Constructor:
public ATextView(*AMenuEnvironment*, *int*, *int*, *int*, *int*, *String*):
Initializes the AButtonView.

Methods:
public void setX(*int*):
Sets the horizontal position of the component.

public void registerInput():
Shorts the input registration.

public void setBackgroundColor(*Color*):
Defined by the subtype component or by abstract creation. Meant to be used on a click event.

ASliderView

public abstract class **ASliderView<E>**
extends ***AMenuComponent***:

The ASliderView is a class that contains a clickable button that moves based on user mouse input and mouse position. Notches will cause the slider to snap to specific locations, the quantity of notches (and therefore snaps) are based on the number of items in the list that this slider will move through.

Fields:
private E e:
The generic type of the elements accepted for this component's arraylist.

protected final AButtonView button:
The slider button.

private final BufferedImage trackImage:
The background track image.

private final BufferedImage notchImage:
The mid-ground notch image.

protected ArrayList<E> values:
The arraylist that contains all elements for the slider to navigate through.

protected int itemCount:
The item count of all elements in the values arraylist.

protected int previous:
The previously selected index of the slider.

protected int current:
The currently chosen index that the slider is on.

private float notchDistance:
The distance between each notch in the slider.

private boolean showNotches:
If the notch images should be displayed within the slider.

Constructor:
public ASliderView(*AMenuEnvironment*, *int*, *int*, *int*, *int*):
Initializes the ASliderView.

Methods:
public abstract void init():
Defined by the encapsulating class. Used to initialize specific content for each individual iteration.

public void moveSliderTo(*int*, *int*):
Moves the slider button to the specific x and y position, and snaps it to the closest notch.

public abstract void doSetting():
Defined by the encapsulating object at instantiation.

public void build():
Builds the slider immediately with these specific details.

public void showNotches(*boolean*):
Sets if the notches should be drawn to the slider.

Subsystem: Models

AEnvironment

public abstract class **AEnvironment**
implements ***IUpdatable***, ***IDrawable***:

The AEnvironment is a superclass of all Environment-based classes. It contains a reference to the EnvironmentsHandler which is used for multiple types of manipulations such as AEnvironment navigation. It contains references to the AKeyController and AMouseController for this particular Environment subtype.There is a reference to the prebuilt Resources, allowing quick access to pre-allocated resources. The AdvancedPlayer is used to control audio streams which remain particular to this Environment.

Fields:
protected javazoom.jl.player.advanced.AdvancedPlayer audioPlayer:
The AudioPlayer object that controls the sound for background audio.

private EnvironmentsHandler parentEnvironmentsHandler:
The parent EnvironmentsHandler.

private AKeyController keyController:
The Environment's Key Controller that's particular to the type of Environment.

private AMouseController mouseController:
The Environment's Key Controller that's particular to the type of Environment.

private Resources resources:
The Resources that exist persistently across all objects.

Methods:
protected void init(*EnvironmentsHandler*, *AControls*):
Initializes with the EnvironmentHandler and the controller used for this particular Environment subtype.

public AKeyController getKeyController():
Retrieves the Environment's AKeyController subtype.

public AMouseController getMouseController():
Retrieves the Environment's AMouseController subtype.

public abstract void startBackgroundAudio():
Should be defined in subclasses to specify what it means to enable the current audio stream. Called primarily during onResume executions. Some subclasses might not need this to execute, so the body may be undefined there.

protected void stopBackgroundAudio():

Called primarily during onExit executions. Some subclasses might not need this to execute, so the body may be overridden as undefined there.

public abstract void reset():
Defined by subclasses. Used to completely default all fields.

public EnvironmentsHandler getParentEnvironmentsHandler():
Retrieves the EnvironmentHandler that contains this object.

public Resources getResources():
Retrieves the Resources pool that this class has access to.

public void setResources(*Resources*):
Sets the Resources locally. Used by any classes contained within or deriving this class.

public void onResume():
Executes when the process deems that the Environment should continue to execute.

public void onExit():
Executes when the process deems that this Environment should cease.

AMenuEnvironment

public abstract class **AMenuEnvironment**
extends ***AEnvironment***
implements ***IDrawable***, ***IUpdatable***:

AMenuEnvironment class is an AEnvironment subtype that controls any Menu Environment and its contained entities. Child subtypes include both MainMenuEnvironment and PauseMenuEnvironment.

Fields:
private final Stack<AMenu> menuStack:
The stack of AMenu items that controls the current menu that's active.

Methods:
public void init(*EnvironmentsHandler*, *MenuControls*):
Initializes the AMenuEnvironment through super calls from the subclass.

public void initPage(*AMenu*):
Initializes the first page with a default AMenu.

public void push(*AMenu*):
Accepts an AMenu subtype and pushes it to the top of a stack of AMenus. The top Menu will always be the one that is active.

public AMenu peek():
Returns the reference to the top-most Menu in the stack.

public AMenu pop():
Removes the top-most Menu in the stack and returns it.

protected int getStackDepth():
Returns the size of the stack

GameEnvironment

public class **GameEnvironment**
extends ***AEnvironment***
implements ***IDrawable***, ***IUpdatable***:

GameEnvironment class is an AEnvironment subtype that controls the Game Environment and its contained entities. It contains the GameControls, Levels, Game HUD, Player Inventory, and all Actors. GameEnviroment class also controls the PauseMenuEnvironment when the GameEnvironment is paused.

Fields:
private GameControls gameControls:
The Game Controls for the Game Environment

private PauseMenuEnvironment pauseMenuEnvironment:
The PauseMenuEnvironment that is used for the pause state

private LevelsList levelsList:
The list of Levels that the user will navigate between.

private PlayerInventory inventory:
The player's inventory of collected items.

private HUDModel hudModel:
The heads-up-display that contains all overlays.

private boolean isPaused:
If the game is paused

private Robot robot:
The Robot which keeps the mouse centered in the window in the unpaused game.

private PlayerAvatar character:
The Player Avatar model

private final ArrayList<AActor> actors:
The list of actors currently active within the current level.

private Queue<AActor> actorsQueue:
The to-add actors that will be added to the list of actors when available.

Methods:

public void init(*EnvironmentsHandler*, *PauseMenuEnvironment*, *GameControls*, *LevelsList*, *HUDModel*, *PlayerInventory*):
Initializes the GameEnvironment with references to pre configured objects.

private void setHUDModel(*HUDModel*):
Sets the preconfigured HUD for the Game Environment

private void setPauseMenuEnvironment(*PauseMenuEnvironment*):
Sets the PauseMenuEnvironment for the paused state of the Game Environment.

public void SetLevelList(*LevelList*) :
The LevelsList that contains all of the Levels in the game

private void setPlayerAvatar(*GameControls*, *LevelsList*):

public void setCurrentLevel(*int*):
Sets the current level by the use of index

private void setPlayerInventory(*PlayerInventory*):
Sets a preconfigured player inventory into the Game Environment

public void setPaused(*boolean*):
Sets the game environment to paused.

public void build(*GameControls*):
Builds the environment with a new Player Avatar and pairs the controls and Levels List directly with the Player.

public void doGameUpdates(*float*):
Updates the objects within the Game state. Controls the game objects, does collisions, updates positions, and updates the Overlay

public void doPauseMenuUpdates(*float*):
Controls the pause state of the Environment. Switches to when game is paused.Allows for use and visuals of pause menu controls

private void doGameControls():
Listens to Game Controller and produces the requested actions from Key inputs for the game state.

public void doPauseMenuControls():
Listens to the Menu Controller and produces the requested actions from Key inputs for the pause state.

public LevelsList getLevelsList():
Gets the LevelsList object which contains all loaded Levels.

private void updateHUD(*float*):
Updates components within the in-game HUD overlay.

private void updateLevel(*float*):

Updates components at the currently active level.

private synchronized void testAddingActors(*float*):
Adds test actors into the level. This is solely used for performance and behavior testing

public void updateActors(*float*):
Updates all actor objects within the current level.

private void detectCollisions(*float*):
Detects the collisions of platforms with actors. Resolves the collisions within the actor and platform objects themselves.

private void insertQueuedActors():
Inserts an actor from the queue into the primary list of actors.

public void queueActor(*AActor*):
Queues an actor into the queue list. Actors are incrementally shifted to the main list when the process is available.

public void addActor(*AActor*):
Adds an actor to the main actors list

public ALevel getCurrentLevel():
Gets the current level that's active

public ACharacter getPlayerAvatar() :
Gets the PlayerAvatar object

public PlayerInventory getPlayerInventory():
Gets the player inventory.

public void update(*float*):
Update method accepts the actual/target ratio to regulate the update rate.

public void draw(*Graphics2D*):
Allows for standard draw calls from a Canvas object.

public void startBackgroundAudio():
Should be defined in subclasses to specify what it means to enable the current audio stream.

public void reset():
Defined by subclasses.

MainMenuEnvironment

public class **MainMenuEnvironment**
extends ***AMenuModel***:

MainMenuEnvironment is a subtype of AMenuEnvironment. It derives behavior and builds on it to create a more expansive version of a Menu environment. It contains behavior that makes it more detailed than the Pause Menu, namely through visual fidelity and reactions

to user interactions, such as with the moving menu background.

Fields:
protected BufferedImage backgroundImage:
The background image of the menu.

Methods:
public void init(*EnvironmentsHandler*, *MenuControls*):
Initializes the MainMenuEnvironment with the parent EnvironmentsHandler and necessary MenuControls

public AMenu getTopPage():
Returns the top menu page of this menu.

public void popToFirst():
Removes the highest AMenus that resides above the default Menu.

public void navigateToLevelSelectPage():
Navigates the user to the Level Select Page.
public void update(float delta):
Update method accepts the actual/target ratio to regulate the update rate.

PauseMenuEnvironment

public class **PauseMenuModel**
extends ***AMenuEnvironment***
implements ***IDrawable***:

PauseMenuEnvironment is a subtype of AMenuEnvironment. It derives behavior from the AMenu class and is contained within the GameEnvironment. This class handles menu-based user interaction from within the Game Environment.

Fields:
protected GameEnvironment gameEnvironment:
The GameEnvironment that this pause menu belongs to.

Methods:
public void init(*EnvironmentsHandler*, *MenuControls*, *GameEnvironment*):
Initializes the MainMenuEnvironment with the parent EnvironmentsHandler and necessary MenuControls.

public void popToFirst():
Pops the top-most AMenus until the landing page is found.

APhysics

public abstract class **APhysics**
implements ***IUpdatable***:

The APhysics class contains data for any in-game object that should contain positional, dimensional, and vector data.

Fields:
protected final float bufferHoriz:
The buffers of the hitbox.

protected final float bufferVert:
The buffers of the hitbox.

protected final float friction:
The multiplier that friction has on the object's motion.

static final float GRAVITY:
The gravity that the object will be affected by.

protected float h:
The width and height of the object.

protected boolean hasGravity:
If the object should be affected by gravity.

protected boolean isFloorCollision:
If the collision on a specific face is made.

protected boolean isMoving:
If this object is moving either horizontally or vertically.

protected boolean isUserControlled:
If the object is being directly controlled by the user.

protected boolean isWallCollisionLeft:
If the collision on a specific face is made.

protected boolean isWallCollisionRight:
If the collision on a specific face is made.

protected final float MAX_VEL_X:
The maximum velocity that the object can move per second.

protected final float MAX_VEL_Y:
The maximum velocity that the object can move per second.

protected float oh:
The original dimension of the object.

protected float ow:
The original dimension of the object.

protected float ox:
The original dimension of the object.

protected float oy:
The original dimension of the object.

protected float vX:

The velocity of the object.

protected float vY:
The velocity of the object.
protected float w:
The width and height of the object.

protected float x:
The position of the object.

protected float y:
The position of the object.

Constructors:
protected APhysics(*float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

Methods:
protected float bottom():
Gets the y+height position.

protected float bottomBufferInner():
Gets the inner bottom boundary based on the vertical boundary and the y+height.

protected float bottomBufferOuter():
Gets the outer bottom boundary based on the vertical boundary and the y+height.

private void calculateGravity(*float*):
Calculates the y velocity under gravity for this object.

float getH():
Gets the height.

float[] getLocation():
Gets the position.

float getW():
Gets the width.

float getX():
Gets the horizontal position.

float getY():
Gets the vertical position.

boolean hasCollision(*AActor*, *float*):
Checks if the object had collisions with the actor in question.

boolean hasCollision(*AActor*, *float*, *boolean*):
Checks if the object had collisions with the actor in question.

boolean isFalling():
Returns if the object has a y velocity of >= 0

protected float left():
Gets the x position

protected float leftBufferInner():
Gets the inner left boundary based on the horizontal boundary and the left side.

protected float leftBufferOuter():
Gets the outer left boundary based on the horizontal boundary and the left side.

void limitVelocity(float):
Limits the velocity if the velocity exceeds the maximum velocity in any direction.

private void resetCollisions():
Resets the state of collisions on the object.

protected float right():
Gets the x + width position.

protected float rightBufferInner():
Gets the inner right boundary based on the horizontal boundary and the right side.

protected float rightBufferOuter():
Gets the outer right boundary based on the horizontal boundary and the right side.

protected void setGravity(*boolean*):
Sets if the object should be affected by gravity.

private void setOriginalPosition(*float*, *float*):
Sets the origin x and y positions.

private void setOriginalSize(*float*, *float*):
Sets the original size of the object.

private void setPosition(*float*, *float*):
Sets the temporary x and y positions

private void setSize(*float*, *float*):
Sets the Width and Height

void setVelocity(*float*, *float*):
Sets the horizontal and vertical velocity.

void setVX(*int*):
Sets the horizontal velocity.

void setVY(*int*):
Sets the vertical velocity

void setX(*float*):
Sets the horizontal position.

void setY(*float*):
Sets the vertical position.

protected float top():
Gets the y position.

void update(*float*):
Updates the object by resetting collisions and then updating the velocity data.

private void updateVelocity(*float*):
Updates the velocity of this object.

AActor

public abstract class **AActor**
extends ***APhysics***
implements ***IDrawable***:

The AActor is the most basic form of entity in the GameEnvironment. Contains a reference to the Resources. Acts as a wrapper, but also contains some data that allows for the appropriate orientations of subtypes.

Fields:
protected AActor.Facing facing:
The enums meant for setting the facing direction.

protected Color color:
The current simple color of the actor's hitbox.

protected AActor.Facing facing:
The current Facing direction.

protected Resources resources:
The persistent Resources.

Constructors:
protected AActor(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

ACharacter

public abstract class **ACharacter**
extends ***APawn***
implements ***IDrawable***:

The prime function of the ACharacter class is that it represents an AActor object that can be controlled directly by the player. This includes, but may not be limited to, the PlayerAvatar

Fields:
protected ACharacter.ActionType actionState:
The current state of the player.

protected CharacterType characterType:
The user-defined chosen character

protected final GameControls controlsModel:
The controls model that this entity will listen to

private boolean isJumpLocked:
If the jump is still executing.

private final int MAX_ALLOWED_JUMP_TIME:
The amount of ticks that a player has to hit the ground or wall after pressing jump before the jump button is no longer recognized as pressed.

private final float MAX_ALLOWED_WALLRIDE_TIME:
The length of time that a player may stick to the wall.

protected HashMap<ACharacter.ActionType,SpriteSheet> spriteSheets:
The hashmap contains all spritesheets particular for specific user actions.

private int time_jump:
The current number of ticks that the player has not hit the ground or wall after pressing jump.

private float time_wallride:
The amount of time that the player has wall ridden for.

Constructors:
ACharacter(*Resources*, *GameControls*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

Methods:
Public void control(*float*):
Called to register the controls to manipulate the entity.

private void doAbilities(*float*):
Calls the process for any programmed abilities.

private void doFloorJumps():
Executes the procedure for a jump from the floor.

public void doJump():
Does the jump procedure.

private void doMovement(*float*):
Uses the controls to manipulate the positional information.

public void doWallJump(*float*, *float*):
Does the procedure for a wall jump.

public SpriteSheet getCurrentSpriteSheet():
Retrieves the current sprite sheet used, based on the action state.

private void lockJumpState(*boolean*):
Locks or Unlocks the character into being in a jump state.

public void reset(*int*):
Resets the data of the character.

private void setActionType():
Checks conditions of the character state to determine the action type of the character.

public void setCharacterType(*ACharacter.CharacterType*):
Sets the type of character, being the male or female character Teo or Melynn.

PlayerAvatar

public class **PlayerAvatar**
extends ***ACharacter***
implements ***IDrawable***, ***IUpdatable***:

The PlayerAvatar is one of the most complex actors. It takes direct control from the GameControls and will react to the controls. Its position controls the Camera's viewport. The PlayerAvatar is the primary actor subtype that all actors check collisions with. The PlayerAvatar will have a direct impact on the game levels, the other actors within a level.

Constructors:
public PlayerAvatar(*Resources*, *GameControls*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object.

Methods:
private void updateSpriteAnimation(*float*):
Updates the image used for the PlayerAvatar. This is obtained through the Spritesheet. The frame of the spritesheet is updated via the delta of the update loop. Some actions use values within to either speed up or slow down the rate at which a sprite changes, or the direction that it changes. The logic uses the action state that the player is currently in and obtains the image from the spritesheet which is correlated with that action.

public void setAction(*ACharacter.ActionType*):
Sets the current action, based on external states and collisions.

AProp

public abstract class **AProp**
extends ***AActor***
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

AProp is an abstract entity type that derives from AActor. This type is simply a wrapper class for Level props. Level props should be allowed to draw to the Map Overlay, which is why the drawToHUD() method is added.

Constructors:
protected AProp(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

Platform

public class **Platform**
extends ***AProp***
implements ***IDrawable***, ***IUpdatable***:

A Platform is a boundary object that acts to create a physical barrier. It is the fundamental piece that allows the other actors to behave as if in a physical world.

Constructors:
public Platform(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object.

ATrigger

public abstract class **ATrigger**
extends ***AProp***
implements ***IDrawable***:

ATrigger is a class which dictates the standard behavior of an object that gets triggered on collision.

Fields:
protected GameEnvironment gameEnvironment:
The parent GameEnvironment.

protected AActor lastActor:
The last actor this trigger affected.

protected boolean canMoveOnCollision:
If the object can move on collision.

protected int MAX_CYCLES:
The maximum number of times this trigger will activate.

protected int currentCycles:

The current number of times the trigger has been activated.

Constructors:
protected ATrigger(*GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*, *boolean*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

Methods:
public abstract void doAction():
doAction defines an action that should be done. It's called, normally, from within the onCollision method when there is a collision with another physical entity.

public void reset()
Resets the current cycle count back to zero.

Spikes

public class **Spikes**
extends ***ATrigger***
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

Spikes are a harmful obstacle. Touching the obstacle immediately kills the player.

Constructors:
public Spikes(*Resources*, *GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*):
Called from the subtypes, this method initializes the object. Also initializes the respective spriteSheet.

Spring

public class **Spring**
extends ***ATrigger***
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

The Spring is a trigger that will set the velocity of the player to something new.

Fields:
private Spring.ActionType actionState:
The current action of the door.

protected HashMap<Spring.ActionType,SpriteSheet> spriteSheets:
The hashmap of spritesheets for the states of the Spring.

Constructors:
public Spring(*Resources*,*GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*, *boolean*, *boolean*):
Called from the subtypes, this method initializes the object. Also initializes the respective spriteSheet.

ALevelCollectible

public abstract class **ACollectibleTrigger**
extends ***ATrigger***:

ACollectibleTrigger is a class built for objects that will be collected upon trigger.

Fields:
protected boolean isActive:
If the trigger can still be activated.

Constructors:
protected ACollectibleTrigger(*Resources*, *GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*, *boolean*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

DoorKey

public class **DoorKey**
extends ***ACollectibleTrigger***
implements ***IDrawable***, ***IUpdatable***:

The DoorKey acts as a collectible. The item is "collected" if the player crosses the boundary into that item. As a collectible, it is added to the Player's Inventory once acquired. If acquired, the object reference is passed to the Inventory, and this item is set to effectively invisible by setting its state to not active.

Constructors:
public DoorKey(*Resources*, *GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*):
Called from the subtypes, this method initializes the object.

ALevel

public abstract class **ALevel**
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

ALevel standardizes the data of a level.

Fields:
protected GameEnvironment gameModel:
The parent GameEnvironment.

private final ParallaxBackground scrollingBackground:
The moving background for the level.

protected int[] startOrigin:
The spawning point of the player.

protected final ArrayList<AProp> levelProps:
The Props that exist within this level.

protected Door door:
The Door that allows the player to exit the level.

protected int keyCount:
The number of keys that exist in the level.

Constructors:
public ALevel(*GameEnvironment*):
Creates a level and obtains reference to the game environment of the level.

Methods:
public abstract void addProp(*AProp*):
Adds a new Prop to the Level.

public ArrayList<AProp> getLevelProps():
Gets the props of the level.

public int[] getCharacterOrigin():
Gets the defined spawn origin point for the player to start in.

public void setStartOrigin(*int*, *int*):
Sets the spawn origin point for the player to start in.

protected void addBackgroundLayer(*BufferedImage*):
Adds a background layer to the parallax background in the level.

public void countKeys():
Counts the number of keys that exist in the level.

public int getKeyCount():
Retrieves the number of keys in the level.

public void build():
Builds the level.

public void reset():
Resets the props of each level.

public void unlockDoor():
Unlocks the door of the level.

AOverlayComponent

public abstract class **AOverlayComponent**
implements ***IUpdatable***, ***IDrawable***:

The AOverlayComponent is the superclass for all HUD overlay components. Contains positional and size data for all components. Also contains a reference to the parent Game Environment.

Fields:
protected GameEnvironment gameEnvironment
The parent GameEnvironment.

protected int x:
The positional coordinate of the component.

protected int y:
The positional coordinate of the component.

protected int w:
The dimensions of the component.

protected int h:
The dimensions of the component.

Methods:
public void init(*GameEnvironment*, *int*, *int*, *int*, *int*):
Initializes the components with positional and size data.

MapOverlay

public class **MapOverlay**
extends ***AOverlayComponent***:

The MapOverlay is drawn to the GameEnvironment's Canvas, which allows for the broader representation of the current level. It shows a depiction of the player position and shows obstacles.

Methods:
public void reset():
Destroys and reconstructs the map image as a blank image.

PlayerStatsOverlay

public class **PlayerStatsOverlay**
extends ***AOverlayComponent***:

The PlayerStatsOverlay is drawn to the GameEnvironment's Canvas, which allows for the representation of the player's inventory.

TimeKeeperOverlay

public class **TimeKeeperOverlay**
extends ***AOverlayComponent***:

The TimeKeeperOverlay is drawn to the GameEnvironment's Canvas, which allows for the representation of the current time the player has spent playing the level.

Resources

public class **Resources**:

First, this class checks for different types of resources along their respective paths. Paths vary between IDE and Jar environments, so those specified within this class respect the paths of both environments. If found, resources are either stored into HashMap lists as objects or as referential paths. Resources are called on by reference, ensuring a singleton instance of the resource existing in memory. This allows for the system to use that resource multiple times without requiring resource overhead.

Fields:
private String PATH_META:
The metadata.json file classpath.

private String path_images:
The classpath for the resource.

private String path_audio:
The classpath for the resource.

private static String path_textFile:
The classpath to all text files.

private String path_fonts:
The classpath for the resource.

private static final Map<String,BufferedImage> imagesFiles:
The Hashmap of String keys obtaining specific BufferedImage references.

private static final Map<String,Clip> audioFiles:
The Hashmap of String keys obtaining specific Audio File classpaths.

private static final Map<String,File> textFiles:
The Hashmap of String keys obtaining specific File references.

private static final Map<String,Font> fontFiles:
The Hashmap of String keys obtaining specific Font references.

Methods:
public void init():
Parses the metadata.json file and registers specified resources locally. The metadata file references all items in the Resources directory, including path names. Those files that are specified within the metadata file are stored referentially or by path depending on the resource type.

private String[] registerFiles(*MetaDataParser*, *String*):
Retrieves file path data held within the MetaDataParser.

private void loadImageFiles(*String*):
Registers all image paths, loads images and stores a reference to the image into a Map.

public BufferedImage loadImageFile(*String*):
Loads the specified image file to see if it exists in the Resource directory. If it exists, the file path is not null.

private void loadAudioFiles(*String*):
Registers all audio paths into a Map

public Clip loadAudioFile(*String*):

Loads the specified audio file to see if it exists in the Resource directory. If it exists, the file path is not null.

private void loadTextFiles():
Registers all text files into a Map.

public File loadTextFile(*String*):
Loads the specified text file to see if it exists in the Resource directory. If it exists, the file is not null. Reads in the text file and stores the data into a temporary file which gets returned.

private void loadFontFiles(*String*):
Registers all Font files into a Map.

public Font loadFontFile(*String*):
Loads the specified Font file to see if it exists in the Resource directory. If it exists, the Font is not null.

public SpriteSheet getSpriteSheet(*String*):
Retrieves a SpriteSheet which is stored in the Map. Uses the references from Text Files and BufferedImage Maps.

public BufferedImage getImage(*String*):
Retrieves a BufferedImage which is stored in the Map

public File getTextFile(*String*)
Retrieves a File which is stored in the Map

public Font getFont(*String*):
Retrieves a Font which is stored in the Map

public javazoom.jl.player.advanced.AdvancedPlayer
playAudio(*String*):
Called externally to start a new concurrent audio thread. Uses stored audio file paths to in-stream the data as an audio file. Sets the file into an AdvancedPlayer object. The AdvancedPlayer object is returned for control over object permanence.

public String toString():
Checks if all files exist within the Map objects. Files do not exist or could not be found if they are null.

ARunnable

public abstract class **ARunnable**
implements ***Runnable***:

The superclass of Render and Update Runnables. This class contains the variables that both subclasses share.

Fields:
protected boolean isRunning:
The runnable is looping.

protected int lastUpdates:
The last number of ticks in the last 1 second cycle.

protected int updates:
The current count of ticks contained in this current 1 second cycle.

Methods:
public void setPaused(*boolean*):
Sets the loop to be paused or not.

RenderRunnable

public class **RenderRunnable**
extends ***ARunnable***:

The Runnable for Render calls. This is an integral piece of all Environments. Contains a continuous loop that dictates the render rate based on a standard tick rate against the current rate. The Current rate is found by determining the number of ticks made in a given second. The ticks are recorded before resetting after a full second. The render rate is normalized by the frame rate maximum, either set manually or by the display device's settings.

Fields:
private ACanvas canvas:
The ACanvas that will be drawn to.

Methods:
public void init(*ACanvas*):
Initializes the subtype Canvas object for this particular render loop.

UpdateRunnable

public class **UpdateRunnable**
extends ***ARunnable***:

The Runnable for Update calls. This is an integral piece of all Environments. Contains a continuous loop that dictates the update rate based on a standard tick rate against the current rate. The Current rate is found by determining the number of ticks made in a given second. The ticks are recorded before resetting after a full second, and the ratio of actual updates vs target updates is found. This ratio, or delta, is passed into the update(float) method of the Environment references. The delta normalizes the updates for all updatable objects in the target environment.

Fields:
private AEnvironment environment:
The AEnvironment that this Runnable updates for.

Methods:
public void init(*AEnvironment*):
Initializes the AEnvironment for this particular Update loop.

AFileReader

public abstract class **AFileReader**:

Fields:
protected File file:
The File that this FileReader has read in.

Constructors:
public AFileReader():
Immediately uses the definition of the abstract read method.

Methods:
public abstract boolean read():
Defined by the inheriting class. Should be dedicated to an IO process for files.

PreferencesParser

public class **PreferencesParser**:

The PreferencesParser accepts the local Preferences file and parses the data within it.The obtained data is used within the Config to initialize the program with default preferences.

Fields:
private File file:
The preferences file that will be parsed through.

private final Config config:
The Configurations that will be registered by the data acquired by the preferences.xml.

Constructors:
public PreferencesParser(*Config*, *File*):
This initializes the PreferencesParser with a reference to the Config and a reference to the preferences.xml file.

Methods:
public boolean parse():
Parses through the preferences.xml through the use of DocumentBuilderFactory.

LevelsList

public class **LevelsList**
implements ***IDrawable***:

The LevelsList contains the list of all Levels that exist for the game. Loads all levels at the same time. Only the current level is updated or drawn.

Fields:
private GameEnvironment gameEnvironment:
The parent Game Environment.

private final ArrayList<ALevel> levels:
The list of all levels available.

private int currentLevel:
The index of the active level.

Methods:
public void init(*GameEnvironment*, *int*):
Initializes the Levels list with the parent GameEnvironment and the starting level.

public void addLevel(*ALevel*):
Adds a new level to the level list.

public boolean setCurrentLevel(*int*):
Sets the index of the current level. Retains the level number if the level does not exist.

public ALevel getCurrentLevel():
Gets the current level.

public boolean navigateNextLevel():
Navigates to the next available level. This is called when the user passes the previous level. Progress is saved to file via this method.
public void reset():
Resets the current level data.

Main

public class **Main**:

Fields:
private static Config config:
The globally used Configurations reference.

private static Resources resources:
The globally used resource files of the classpath.

private static SaveData saveData:
The persistently used save data.

private static SaveFileRW saveFileRW:
The SaveFileRW that the SaveData uses persistently.

private static EnvironmentsHandler environmentsHandler:
The persistent EnvironmentsHandler.

private static MapOverlay mapOverlay:
The persistent PlayerStatsOverlay.

private static TimeKeeperOverlay timeKeeperOverlay:
The persistent TimeKeeperOverlay.

private static HUDModel hudModel:
The persistent HUDModel.

private static PlayerInventory inventory:
The persistent PlayerInventory.

private static GameControls gameControlsModel:
The persistent GameControls.

private static MenuControls menuControlsModel:
The persistent MenuControls.

private static GameEnvironment gameEnvironment:
The persistent GameEnvironment.

private static PauseMenuEnvironment pauseMenuModel:
The persistent PauseMenuEnvironment.

private static UpdateRunnable gameUpdateRunnable:
The persistent UpdateRunnable.

private static RenderRunnable gameRenderRunnable:
The persistent RenderRunnable.

private static UpdateRunnable menuUpdateRunnable:
The persistent UpdateRunnable.

private static RenderRunnable menuRenderRunnable:
The persistent RenderRunnable.

private static LevelsList levelsListModel:
The persistent LevelsList.

private static EnvironmentCanvas<MainMenuEnvironment> menuCanvas:
The persistent EnvironmentCanvas for the MainMenuEnvironment.

private static EnvironmentCanvas<GameEnvironment> gameCanvas:
The persistent EnvironmentCanvas for the GameEnvironment.

private static MainWindow window:
The persistent MainWindow reference.

Methods:
public static void main(*String[]*):
The starting point of the program. First loads all assets from file into a resource pool for later use and ease of access. Then Creates all permanent static variables for protection against null pointer exceptions. Then initializes all variables with their necessary reference objects. Assists in MVC cross-object communication.

public static void loadAssets():
Handles the loading of all assets from files into a resource pool for later use and ease of access.

public static void create():
Creates all permanent static variables for protection against null pointer exceptions.

public static void init():
Handles the initializes all variables with their necessary reference objects. Assists in MVC cross-object referencing.

Sprite

public class **Sprite**:

Fields:
private final String filename:
The Sprite frame name.

private final int[] frame:
The frame size.

private final int[] spriteSourceSize:
The frame size.

private final int[] sourceSize:
The frame size.

private final boolean rotated:
A modification of the frame.

private final boolean trimmed:
A modification of the frame.

private int duration:
The duration that the frame should last for in milliseconds.

private float[] scaledPos:
The scaled position relative to the max sprite size.

private float[] scaledSize:
The scaled size relative to the max sprite size.

Constructors:
public Sprite(*String*, *int*, *boolean*, *boolean*, *int*, *int*, *int*):
Sets the data pulled from the SpriteSheet Json file by means of the SpriteSheetParser.

Methods:
public String getName():
Gets the name of the frame.

public BufferedImage getSubImage(*BufferedImage*):
Gets the frame from the master sprite sheet image.

public int getDuration():
Gets the frame duration.

public boolean isTrimmed():
Gets if the frame is trimmed.

public int[] getSize():
Gets the actual size of the frame.

public void setScaledSize(*float[]*):
Gets the scaled size against the largest width frame and the largest height frame.

public void setScaledPos(*float[]*):
Gets the scaled position vs the largest frame width and largest frame height.

public float[] getScaledSize():
Gets the scale of the frame in relation to the largest frame.

public int[] getPosition():
Gets the scaled position of the frame in relation to the max frame size.

SpriteSheet

public class **SpriteSheet**
implements ***IUpdatable***:

SpriteSheet allows for the compilation of individual Sprite data into a list. Gives access to animations. Allows for animation control of the sprites.

Fields:
private BufferedImage referenceImage:
The spritesheet image that sprites will be sampled from.

private ArrayList<Sprite> frames:
All frame data that exist for this sprite sheet.

private final int[] largestSize:
The largest x and y dimension of all considered frames.

private float[] frameScale:
The frame scale of the largestSize against the max size.

private int currentFrame:
The current index information of the animation.

private int ticks:
The current index information of the animation.

private boolean loopOnLastFrame:
If the animation should restart at the first frame when the last frame is finished.

Constructors:

public SpriteSheet(*ArrayList<Sprite>*):
Creates a spritesheet based on the sprite arraylist passed.

Methods:
public void setReferenceImage(*BufferedImage*):
Sets the Reference image of the spritesheet.

public void incrementFrame(*int*):
Increments the current frame by the increment amount.

public void setFrame(*int*):
Sets the current frame to the index provided. Cycles the frame to 0 only if the animation is set to loop.

public boolean isLastFrame():
Checks if the current frame is the last frame.

public SpriteSheet setLoopOnLast(*boolean*):
Sets if the animation should loop after the last frame.

public float getPercentCompleted():
Gets the animation cycle percent completed.

public SpriteSheet setFrameScale(*float*, *float*):
Records the framescale and sets the framescale into each frame.

public float[] getFrameScale():
Gets the scale of the container vs the largest frame size.

public float[] getCurrentFrameSize():
The current frame's scaled size.

public int[] getCurrentFramePos():
Gets the current frame's position.

public int[] getLargestSize():
Gets the largest dimensions of the largest dimensions across all frames.

public boolean isTrimmed():
Checks if the current frame is trimmed.

public void draw(*Graphics*, *int*, *int*, *int*, *int*):
Draws the Sprite image, pulled as a sub image from the Spritesheet.

public void reset():
Resets the spritesheet data.

SpritesheetParser

public class **SpriteSheetParser**:

The SpriteSheetParser accepts a Json file that contains data that's particular to the output produced by a Spritesheet program

called "Aseprite". Aseprite program allows a user to design animations, and outputs the animation as an image of all frames, conjoined side by side into one image. This program takes the Json file and delimits the frame data. This class parses that Json file and stores each Sprite's data into Sprite objects that are then stored into a SpriteSheet object. That SpriteSheet data is then used when a spritesheet is requested from the Resources.

Fields:
private SpriteSheet spriteSheet:
The Spritesheet that shit parser will write to.

private final String jsonFile:
The path of the json file for this SpriteSheet.

Constructor:
public SpriteSheetParser(*String*):
This initializes the SpriteSheetParser with a Json file to be parsed. It then parses that sheet.

Methods:
private SpriteSheet processSpriteSheet():
Processes the Json file, stores the data into individual Sprites, and stores the Sprites into a SpriteSheet.

public SpriteSheet getSpriteSheet():
Gets the SpriteSheets which has been given data retrieved from a Json containing spritesheet information.

PlayerInventory

public class **PlayerInventory**:

The PlayerInventory contains relevant ad-hoc information about the player's collectibles status within the current level.

Fields:
protected ArrayList<ALevelCollectible> collectibles:
The collectibles that the player has acquired from the level.

Methods:
public void addCollectible(*ALevelCollectible*):
When the player collects a Collectible item, the item is added to the inventory. These are later compared against their type to determine the quantity of each or the effect of each.

public int getKeyCount():
Used to obtain the number of Keys that the player has acquired.

public void reset():
Removes all of the current collectibles from the collection.

Camera

public class **Camera:**

Camera class contains static variables and methods which allow for the manipulation of viewport-based data. Viewport data is used in all Environments where there is position-based renders. In the Game state, the player position will control the camera offset. In the Menu state, the mouse position will control the camera offset. Camera zooms in or out to act to increase or decrease the scaling of in-game objects.

Fields:
public static final float DEFAULT_ZOOM_LEVEL:
The default level of zoom. Where the target resolves to if reset.

public static float zoomLevel:
The current zoom level.

public static float zoomTarget:
The target zoom level.

public static float targx:
The target camera coordinates.

public static float targy:
The target camera coordinates.

public static float camX:
The current chase camera coordinates.

public static float camY:
The current chase camera coordinates.

public static float mapX:
The current non-chase camera coordinates.

public static float mapY:
The current non-chase camera coordinates.

public static float DEFAULT_ZOOM_ACCELERATION:
The default acceleration for the camera.

public static float DEFAULT_PAN_ACCELERATION:
The default acceleration for the camera.

public static float acceleration_pan:
The current acceleration for the camera.

public static float acceleration_zoom:
The current acceleration for the camera.

Constructors:
public Camera():
Initializes the Camera to the standard position and scale.

Methods:

public static void moveTo(*float*, *float*):
This accepts a default-scale-relative location in the horizontal and vertical axis'. The following global variables manipulate scale and position to reflect new translations.

public static void zoomTo():
Acts to zoom the camera to the target zoom level.

public static void reset():
Resets the positional data of the camera.

Config

public class **Config**:

The Configuration class deals with essential system data. Heavily used for scaling of all rendered elements, managing render and update rates, and determining window types.

Fields:
private static String opSysName:
The name of the Operating System.

public static String jarPath:
The system path of the jar.

private static final DisplayInfo displayInfo:
The DisplayInfo of the system.

private static Config.WindowType window_type:
The type of frame for this window.

public static int DEFAULT_WINDOW_WIDTH:
The default window dimension that all render scales compare to.

public static int DEFAULT_WINDOW_HEIGHT:
The default window dimension that all render scales compare to.

public static int window_width_selected:
The user requested window width.

public static int window_height_selected:
The user requested window width.

public static int window_width_actual:
The actual window width. Is restricted by the screen dimensions.

public static int window_height_actual:
The actual window width. Is restricted by the screen dimensions.

public static float scaledW:
The scaled dimensions of the window against the DEFAULT_WINDOW dimensions.

public static float scaledH:

The scaled dimensions of the window against the DEFAULT_WINDOW dimensions.

public static float scaledW_zoom:
The scaled dimensions, with camera zoom considered, compared against the DEFAULT_WINDOW dimensions.

public static float scaledH_zoom:
The scaled dimensions, with camera zoom considered, compared against the DEFAULT_WINDOW dimensions.

public static short GAME_UPDATE_RATE:
The default tick rate of the environments.

public static short FRAME_RATE_DEFAULT:
The default target framerate for the renders.

public static short frameRate:
The actual framerate for the renders.

Methods:
public static void registerSystemInfo():
Looks at the system configurations for operating system type and finds the JAR path.

public void calcResolutionScale():
Called after the system obtains the current Window. Records the scale factor of the current window size against the default window size. Allows for accurate scaled renders of the environment's elements.

public void setWindowWidthSelected(*int*):
The desired dimensions of the frame size. Used against the default dimensions to obtain proper scaling. This may become the actual dimension, unless there are conflicts.

public int getWindowWidthSelected():
The desired dimensions of the frame size. Used against the default dimensions to obtain proper scaling. This may become the actual dimension, unless there are conflicts.

public void setWindowHeightSelected(*int*):
The desired dimensions of the frame size. Used against the default dimensions to obtain proper scaling. This may become the actual dimension, unless there are conflicts.

public int getWindowHeightSelected():
The desired dimensions of the frame size. Used against the default dimensions to obtain proper scaling. This may become the actual dimension, unless there are conflicts.
public void setWindowWidthActual(*int*):
The actual dimensions of the frame size. Used against the default dimensions to obtain proper scaling.

public void setWindowHeightActual(*int*):
The actual dimensions of the frame size. Used against the default dimensions to obtain proper scaling.

public void setWindowWidthDefault(*int*):
The default dimensions standardize the scale factor between default and user-configured frame size.

public void setWindowHeightDefault(*int*):
The default dimensions standardize the scale factor between default and user-configured frame size

public void setWindowType(*int*):
Accepts the desired WindowType ordinal. This will be used by the AWindow for designing the style of frame requested.

public static void setWindowType(*Config.WindowType*):
Accepts the desired WindowType. This will be used by the AWindow for designing the style of frame requested.

public ConfigData.WindowType getWindowType():
This will be used by the AWindow uses WindowType for designing the style of frame requested.

public void setGameUpdateRate(*short*):
Used to standardize variations in tickrate, where the ratio gets shared among all updatable entities to normalize movement

public void setFrameRate(*short*):
This is the framerate that's controlled by the config and has the option to be changed by the user.

public void setFrameRateDefault(*short*):
The standardized rate of draw calls set by the config file. This will be the target frame rate unless there's a higher limit set by the default display or if the user changes their preferences.

public void post():
The wrapper method used to calibrate render scaling during first-time setup.

APawn

public abstract class **APawn**
extends ***AActor***
implements ***IDrawable***, ***IUpdatable***:

A Simple Game Object which carries the ability to move. It is inherited by classes whose objects must physically move within the world.

Constructors:
protected APawn(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):

Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

APropTrigger

public abstract class **AProp**
extends ***AActor***
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

APropTriggers are used to standardize the functionality of objects which react to collisions.

Constructors:
protected AProp(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

AReactProp

public abstract class **AReactProp**
extends ***AProp***
implements ***IDrawable***:

The ReactProp ensures that a prop will contain a reaction method as well as the standard Trigger methods.

Fields:
protected GameEnvironment gameEnvironment:
The parent GameEnvironment.

protected boolean canMoveOnCollision:
If the prop should move upon collision.

protected int currentCycles:
The number of times this prop has been activated.

Constructors:
Protected AReactProp(*Resources*, *GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*, *boolean*, *boolean*):
Called from the subtypes, this method initializes the object with position and size relative to the default dimensions.

Methods:
public abstract void onReact():
Defined by the inherited class.

public void reset():
Resets the cycles.

DisplayInfo

public class **DisplayInfo**:

DisplayInfo is the configuration representative for the default display. It accepts custom scaling settings done within the system

settings of the computer, allowing for scale factor changes within the Config.

Fields:
public static float SYS_SCALEX:
The system's extra scale factor of all GUI components.

public static float SYS_SCALEY:
The system's extra scale factor of all GUI components.

private ArrayList<Dimension> windowDimensions:
List of all screen dimensions.

Constructors:
public DisplayInfo():
Uses a temporary JFrame to obtain graphics configurations. Graphics configurations are then used to determine the custom scale of the system.

Door

public class **Door**
extends ***APropTrigger***
implements ***IDrawable***, ***IHUDDrawable***, ***IUpdatable***:

The Door is meant to be the gate by which the user will collide with in order to progress to the next level. Should the Door be locked, the player must collect DoorKeys to unlock it. Once unlocked, the player will walk into a boundary area that triggers the door open. When opened, the door's state will become 'open'. From there, the player may proceed to trigger the door event.

Fields:
private Door.State state:
The current state of the door.

private final Door.Type type:
The action of the Door.

protected HashMap<Door.Type,SpriteSheet> spriteSheets:
The Hashmap containing all spritesheets relative to a Door.

Constructors:
public Door(*Resources*, *GameEnvironment*, *float*, *float*, *float*, *float*, *float*, *float*, *int*, *boolean*, *boolean*):
Called from the subtypes, this method initializes the object.

Methods:
public void onReceive():
This is called when another ATrigger subtype defines their action to call this action. In other words, the onReceive method is beholden to another trigger. This event will open the door if it is closed.

public void doAction():

doAction defines an action that should be done.

public void unlock():
Unlocks the door.

EnvironmentsHandler

public class **EnvironmentsHandler**:

Contains references to all AEnvironments, and holds references to each of their ACanvases and ARunnables.

Fields:
private MainWindow parentWindow:
The main window frame.

private SaveData saveData:
The save data with previously saved information and current information.

private EnvironmentsHandler.EnvironmentType currentEnvironment:
The current environment that the user is in.

private final ArrayList<AEnvironment> environments:
The list of all environments available.

private final ArrayList<ACanvas> canvases:
The list of all canvases available

private final ArrayList<ARunnable> updateRunnables:
The list of all update runnables available

private final ArrayList<ARunnable> renderRunnables:
The list of all render runnables available

private Thread updatesThread:
The Updates thread that all environments' update runnables will use.

private Thread rendersThread:
The Render thread that all environments' render runnables will use.

Methods:
public void init(*MainWindow*, *SaveData*):
Initializes the EnvironmentsHandler.

public void addEnvironmentPair (*AEnvironment*, *ACanvas*, *ARunnable*, *ARunnable*):
Adds a new AEnvironment subtype to the list of AEnvironments.

public void setCurrentEnvironmentType
(*EnvironmentsHandler.EnvironmentType*):
Sets the current EnvironmentType.

public EnvironmentsHandler swapToEnvironment
(*EnvironmentsHandler.EnvironmentType*, *boolean*):
Handles the procedure of switching from one AEnvironment to another. AEnvironments do not all need to be reset due to some AEnvironments existing at certain points. Therefore, a reset condition is passed.

public AEnvironment getCurrentEnvironment():
Returns the active AEnvironment.

public GameEnvironment getGameEnvironment():
Returns the GameEnvironment.

public MainMenuEnvironment getMenuEnvironment():
Returns the MainMenuEnvironment.

public ACanvas getCurrentCanvas():
Returns the active AEnvironment's canvas.

public Runnable getCurrentUpdateRunnable():
Returns the active AEnvironment’s updates-based ARunnable.

public Runnable getCurrentRenderRunnable():
Returns the active AEnvironment’s render-based ARunnable.

public void applyEnvironment():
Calls the applyEnvironment method with the reset default set as true.

public void applyEnvironment(*boolean*):
Applies the current AEnvironment to the Window. Restarts the current AEnvironment’s Thread if necessary.

public void pauseThreads():
Pauses the current AEnvironment’s ARunnable.

public void initThreads():
Destroys active threads and creates new Threads using the current AEnvironment’s ARunnables.

public void rebuildWindow():
Rebuilds the window with the appropriate window dimensions and WindowType.

HUDModel

public class **HUDModel**
implements ***IDrawable***, ***IUpdatable***:

The HUDModel is the handler class for all AOverlayComponents of the GameEnvironment. It acts to update and draw all AOverlayComponents as needed.

Fields:
protected GameEnvironment gameEnvironment:

The parent GameEnvironment.

private MapOverlay map:
The local MapOverlay.

private PlayerStatsOverlay stats:
The local PlayerStatsOverlay.

private TimeKeeperOverlay timer:
The local TimeKeeperOverlay.

Methods:
public void init(*GameEnvironment*, *MapOverlay*, *PlayerStatsOverlay*, *TimeKeeperOverlay*)
Initialized the HUDModel with pre-created AOverlayComponent objects.

public void reset():
Acts to reset the map as-needed. Normally called during the reset of the parent GameEnvironment.

IDrawable

public interface **IDrawable**:

Methods:
void draw(*Graphics2D*):
Allows for standard draw calls from a Canvas object.

IHUDDrawable

public interface **IHUDDrawable**:

IHUDDrawable is used by HUD Overlay classes which require draw calls for a render thread.

Methods:
void update(*float*):
Allows for the user to specify a draw call for an Overlay component.

MetaDataParser

public class **MetaDataParser**:

MetaDataParser class accepts the metadata.json file. This file is used to list all local files within the classpath, register them to a hashmap, and allow this data to be transferred into the Resources for further processing and resource allocation.

Fields:
private final String jsonFile:
The list of background layers.

private HashMap<String, String> paths:
The resource paths of all data types.

private HashMap<String,ArrayList<String>> fileNames:
The arraylist of file names based on the file type.

Constructors:
public MetaDataParser(*String*):
Creates a new MetaDataParser and stores the path of the metadata.json file.

Methods:
public void init():
Reads the metadata.json file and uses Gson to parse through the content.

public String getPath(*String*):
Gets the path of the specific Resource type.

public ArrayList<String> getFileNames(*String*):
Gets the file names of the type of resource that is requested.

private void registerElement(*com.google.gson.JsonObject*, *String*):
Registers the Element of the specified attributes passed.

ParallaxBackground

public class **ParallaxBackground**
implements ***IDrawable***:

The ParallaxBackground contains references to images. These images act as layers in the moving background of the level. The level background will move, and each layer will move at separate rates, respective to their layer number. The closer the layer is to the player, the faster the layers will move.

Fields:
private final ArrayList<ParallaxBackground.Layer> layers:
The list of background layers.

private float moveScale:
The standard scale of movement for the front-most image.

Methods:
public void addLayer(*BufferedImage*):
Adds a new Image layer to the list of background images.

Particle

public class **Particle**
extends ***APawn***
implements ***IDrawable***, ***IUpdatable***:

A Particle is a Pawn that accepts an initial velocity and position. It follows its vector and does nothing else.

Constructors:

public Particle(*Resources*, *float*, *float*, *float*, *float*, *float*, *float*, *boolean*):
Called from the subtypes, this method initializes the object.

SaveData

public class **SaveData**:

The data obtained from the users previously written save file. Save files are simple, as they are only manipulated to store and retrieve the user's last completed level. The Save Data will automatically update if a level is completed. Progress is reset if the user chooses to start a New Game. Progress is written to file either at the next available time whenever the user's progress is updated or when the user closes the program. Non-graceful termination may lead to corrupted save file or lost progress from the previous execution.

Fields:

private SaveFileRW saveFileRW:
SaveFileRW parsing process.

private int lastCompletedLevel:
The character type used for this run.

private int lastCompletedLevel:
The last level that the user has completed.

Methods:

public void init(*SaveFileRW*):
Initializes the progress to default values.

public SaveData setLevelProgress(*int*):
Sets the user's level progress.

public int getLevelProgress():
Gets the user's last completed level.

public void save():
Requests to record the SaveData to file.

SaveFileRW

public class **SaveFileRW**:

SaveFileRW is granted access to the SaveData that's held in memory and updated internally, and processes that data to Json format and writes it to a persistent external file. If the file is not found,

the file is created. The external save file is always created at the path of the JAR.

Methods:

public void init(*SaveData*, *String*):
Construct the SaveFileRW with persistent Files.

public void createNewSaveFile():
Creates a new Save File if the save file does not exist. The save file is always created at the same system path of the JAR.

public boolean savetoFile():
Serializes the save data to the save file.

private boolean serialize():
Writes SaveData object to File. Validate File by deserializing the file and checking the data.

public boolean deserialize():
Read from File and records to SaveData object.