Game Engine Structure

This engine is designed to work best with premade image sequences with transparency that will be used for animation. I based most of the function names off of Flash Actionscript. The game engine operates in the Model-View-Controller pattern of user interaction. That means that all the components will be split into these three categories. Model contains all of the information about the game including lists of objects, their positions, attributes, and much more. View contains the code that actually draws the images to the screen. Controller handles all of the “number crunching” and game logic that makes the game run.

To make a basic game, you only need to work within three files:

- model/Model.java – This file is where you will declare the variables, game objects, and any data structures that you want in your game. In this file is also the startGame() method which is used to instantiate any objects and more importantly create movie clips and add them to the display list (more on this later).
- controller/FrameHandler.java – Inside this file is the main game loop that executes at 30 frames per second (or the rate you choose to set). You will edit the enterFrame() method. This will hold the bulk of your code and handles everything from telling objects to move and checking for collisions to checking if the game should be over.
- controller/MouseHandler.java – This file will contain all code that you want to occur during mouse events such as when the player presses down or releases the mouse.

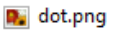
The main method is located in controller/Controller.java.

The MovieClip Class

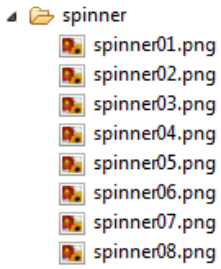
This is the most important class in the game engine. It contains all of the images in animation sequences and the position, rotation and scale of the objects. The movie clip is capable of loading in many folders of images, organizing them into animations and then telling the view what specific frames to draw, where, and with what transformations. For example, you could load in a “player” folder, and it would split up running, walking and standing sequences into separate animations to switch between. It contains easy tools for switching between frames and full animation sequences. It also contains a number of functions to ease collision detection and rotation tasks.

You can declare a movie clip in three ways:

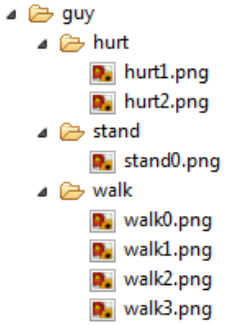
```
new MovieClip(”dot.png”)
```




```
new MovieClip(”spinner”)
```



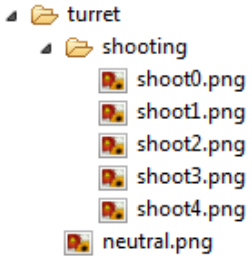

```
new MovieClip(”guy”, true)
```



1. If you only want a static image, you just pass in the relative/absolute path to that image.
2. If you want a single sequence of images, you pass in the relative/absolute path to the folder.
3. If you want a movie clip that will be able to switch between multiple animations, you should pass in the relative/absolute path to the folder, along with true. Including true lets the game engine know to look out for nested folders. You can only nest one level.

Notes:

- The animation will sequence the images alphabetically. You can number them however you want, just make sure that they are sequenced alphabetically.
- If it is only a static image and not a folder, make sure to include the file extension (example: .png).
- Nested folders can mix folders of images and static images. See below:



Using the MovieClip Class

Here is a basic example of how you would make a ball slide across the screen:

model/model.java

```
.
.
.
public MovieClip ball;

public void startGame(){
    ball = new MovieClip("ball.png");
    ball.x = 200;
    ball.y = 200;
    addChild(ball);
}
.
.
.
```

Controller/FrameHandler.java

```
.
.
.
private void enterFrame(){
    model.ball.x += 5;
}
.
.
.
```

Just with these small additions, it will load the image into the game, give it position variables and then move it 5 pixels every frame. The repainting is all handled by the View so you do not need to worry about calling it. The function addChild belongs to the Model and adds the movie clip to the display list. Display lists will be covered more thoroughly in the next section. Notice how since enterFrame() is not in the model file, it must specify "model." before objects.

Here are the most common commands concerning movie clips (see the interface file for the full list):

These variables control the position and transformation and can be directly changed.

`int x` – The horizontal position in pixels (0 is left side)
`int y` – The vertical position in pixels (0 is top)
`double rotation` – The rotation clockwise in radians
`double scaleX, scaleY` – The scales with 1 as full size. 0.5 = half, 2 = double

The following are functions used to control the movie clips:

`void setOrigin(int ox, int oy)` – This method is usually called directly after creating a movie clip. You put in the coordinates of the center of the image. This point will be the point of the picture that should be located at x,y when you set the position variables. This spot is also where the object will rotate and stretch around.

`Point getDimensions ()` - This will return the pixel dimensions of the current playing animation. It will return it as a point: (width,height).

`void play()` - If the animation is paused, or playing in reverse, this will cause it to play forwards.

`void stop()` - This will pause the current animation.

`void gotoAndPlay(int n)` - This will cause the current animation to go to a specific frame, then play.

`void gotoAndStop(int n)` – This will cause the current animation to go to a specific frame, then stop.

`void stopAtEnd()` - This will tell the movie clip that when it reaches the end of the current animation, it should pause.

`void setDirection(Direction dir)` – This will cause the current animations (and any animations you switch to) to play in a certain direction or pause. The options are Direction.FORWARD, Direction.BACKWARD, and Direction.PAUSED.

If you used the third type of constructor, you have included multiple animations/images for the movie clip to switch between. Each will be automatically named based on the folder/image. Remember that if it was a static image, it will include the file extension.

`void swapAndRestart(String name)` – This will switch the movie clip to play a different animation. The animation will have been automatically named based on the folder. In the turret example, I could write swapAndRestart("shooting"), or swapAndRestart("neutral.png"). This function will begin playing that animation at its start and in the forward direction.

`void swapAndResume(String name)` – This will switch the animation similarly to swapAndRestart except that it will not affect Direction. If it's playing backwards, it will continue playing backwards, paused will stay paused etc.

`String getCurrentAnim()` - This will return a string representing the current animation playing.

There are four functions used for collision detection. A collision can be between the current movie clip and another movie clip, or between the current movie clip and a point. Each collision can be a rectangular collision (based on the image's bounding box), or it can be given a specific shape (the only one implemented currently is "circle").

```
boolean hitTest(MovieClip mc)
```

Here is a more complex example of the movie clip features:

model/Model.java

```
.
.
.
public MovieClip player;
public MovieClip spinner;

public void startGame(){
    player = new MovieClip("guy",true);
    player.setOrigin(24,24);
    addChild(player);
```

```
    spinner = new MovieClip("spinner");
    spinner.setDirection(Direction.BACKWARD);
    spinner.x = 300;
    spinner.y = 300;
    addChild(spinner);
}
.
.
.
```

controller/FrameHandler.java

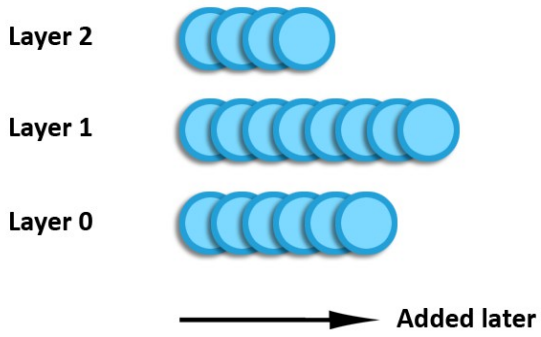
```
.
.
.
private void enterFrame(){
    model.player.x = mouse.mouseX();
    model.player.y = mouse.mouseY();

    if (model.player.hitTest(model.spinner,"circle")){
        model.player.swapAndResume("hurt");
    }else{
        model.player.swapAndResume("walk");
    }
}
.
.
.
```

This will create two objects, a player and spinner. The player can be moved with the mouse, and while he is overlapping the spinner, he will play the hurt animation. The player is 48x48 pixels so the setOrigin will ensure that the player's center will be located at the mouse.

The Display List

Until now you have probably been wondering how everything is drawn properly to the screen and what the addChild command really does. Everytime addChild is called with a movie clip, it is adding this movie clip to an ordered list inside model called the display list. **(Important: you only need to call addChild once, not for every frame)** The order is preserved in the display list so that things that are added earlier will be drawn earlier and therefore be "under" other things drawn on the screen. Of course there are ways to control the depth of all the objects after you create them.

This can be done with the swapDepths command which will take two movie clips and switch their ordering. But you can get even more control using layers. By default there are 3 layers numbered 0,1,2 with 0 being the lowest layer. You can specify which layer to add an object to by saying something like addChild(ball,1). If no number is given (like in the earlier examples) it will be added to layer 0 by default. Here is a graphical explanation of the way the lists and layers interact:



Depth will be determined by both the order the movie clip was added and by the layer its on. Layers take precedence so everything on layer 1 will be above everything on layer 0, for example.

Here are the most common functions needed for dealing with the display list and layers.

`void addChild(MovieClip mc, int layer)` – This will add a movie clip to the end of the list on its appropriate layer. You can leave out layer and just call addChild(MovieClip mc) if you do not care about layers.

`void moveLayer(MovieClip mc, int layer)` – This will take a movie clip off of a layer, and put it at the last position of a given layer (front-most visibly).

`void removeChild(MovieClip mc)` – This will find the layer a movie clip is on and remove it.

`void swapChildren(MovieClip mc1, MovieClip mc2)` – This will swap the depths of two movie clips; the movie clips can be on the same layer or different layers. In both cases, the exact depths within each list will be preserved.

`void bringToFront(MovieClip mc1)` – This will bring a movie clip to the last position of its layer (front-most visibly). There is also an accompanying sendToBack command.

All of these functions belong to the model, but can be called from anywhere at anytime[1].

1 The only place you cannot call these is in getNextFrame in MovieClip because changing the display list while painting will disrupt the iterator which is drawing the objects to the screen sequentially by calling getNextFrame.

Keyboard and Mouse

Both the keyboard and mouse have specific handlers that take care of a lot of behind-the-scenes work leaving you with some helpful functions.

Keyboard

You will probably never have to alter the keyboard handler, instead the most useful functionality is passed through the isDown function. Anywhere inside the enterFrame function you can call keys.isDown(keycode) and insert the number for the ASCII key code for that character.

For example:

controller/FrameHandler.java

```
private void enterFrame(){
    if(keys.isDown(32)){
        model.player.x += 5;
    }
}
```

This will cause the player to move right continuously as you hold space.

Do not be tempted to try to override the keyPressed function inside the handler for code that you only want to execute *only once* for every key press. Doing this in the keyPressed method will cause it to behave like when you are typing: if you hold a key, it will actuate once, pause, then actuate repeatedly as the key is held down. If you desire *once per press* functionality, make use of the keyJustPressed function.

controller/FrameHandler.java

```
private void enterFrame(){
    if(keys.keyJustPressed(32)){
        System.out.println("space");
    }
}
```

It is important to note that keyJustPressed returns true if the state of the button has changed since the last time step. There is also a function keyJustReleased.

Also included for the keyboard is a helper function for arrowkey/aswd movement. If you call keys.arrowVector() or keys.aswdVector() it will return a unit vector representing the 8-way directions that could occur by holding combinations of keys. Since a unit vector is magnitude 1, just multiply it by the move speed to get a motion vector. You can apply simple movement for example by writing:

```
model.player.x += (int) (keys.arrowVector().getX()*10.0);
model.player.y += (int) (keys.arrowVector().getY()*10.0);
```

Mouse

Likewise for the mouse, you can call mouse.leftDown() or mouse.rightDown() inside the enterFrame() function to tell what buttons are pressed down.

In addition you can use the functions mouseX, mouseY, isOnScreen, hideMouse, and showMouse which behave as expected.

Unlike the keys, it is more common to desire specific mouse listener events such as press, release, click etc. For these, you can edit the MouseHandler.java file.

Inside this file you are free to override any of the mouse action listener functions. Just make sure to follow 2 rules:

1. Include synchronized(model) { … } around the contents of each function to make sure there are no concurrency issues between the listener threads and the game loop thread.
2. Make sure to call the super method (super.mousePressed(m) for mousePressed for example). This will prevent the new functions from missing any functionality of the mouse that already exists.

Advanced Features

MiniTimer

I've included the mini timer to make intervaled frame events easier to handle. This timer does make use of any threads, so you have to keep it updated every frame for it to function correctly, but it is still time-saving in the big picture.

You create the timer by giving it a frame interval and number of repeats (0 for infinite looping).

```
MiniTimer spawnCoin = new MiniTimer(30,0);
```

then in enterFrame:

```
if(spawnCoin.isReady()){
    MovieClip coin = new MovieClip("coinspin");
    coin.x = (int) (Math.random()*300);
    coin.y = (int) (Math.random()*300);
    model.addChild(coin);
}
```

By calling isReady, you are simultaneously keeping the timer updated by incrementing its internal timer, and getting a boolean value back that will be true if the timer fires.

Derivative

I created this class to help with events that should only happen when a certain state *changes*. For example just when the player *enters* an area, but not *while* he is in the area. Given a stream of boolean input, it will return the “derivative” in a stream of output. Here is an example of input and output for a 16 frame period of time:

```
    F, F, F, F, F, F, T, T, T, T, F, T, F, F, F, F
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0,-1, 1,-1, 0, 0, 0
```

It will return 1 if the value has just changed from false to true, -1 if true to false, and 0 if the current frame is the same as the last. Note that the first output will always be 0.

The constructor is called in model with no input, then you use getDelta(input) during each time step in enterFrame.

Heres an example of it being used for area leaving:

```
if(areaDeriv.getDelta(model.player.hitTest(model.area)) == -1){
    //do something
}
```

An important thing to note is that getDelta returns the difference since the last getDelta function call, not the last time-step, so if you want check if something has changed since the last frame, be sure to call it exactly once per time step.