**Milestone II Design Doc**

**Design Objective:** Allowing developers to create new fighters and define their animation combo moves without having to draw extra sprite images.

**Design Problem:** 1) Coordinating movement of fighter body parts

2) Define animation for any arbitrary limb of a fighter

**---------------------------------PART I INTRODUCTION --------------------------------------**

**Simple Use Case:** Create a fighter made of a single sprite image, located in the center of canvas, moves up/down/left/right

**Step 1)** Create a LimbNode

BufferedImage fighterImg = ImageUtil.loadImage(“….”);

LimbNode fighter = new LimbNode(“fighter”, fighterImg, this.getWidth()/2, this.getHeight()/2);

**Step 2)** place LimbNode in a BodyTree

BodyTree tree = new BodyTree(fighter);

**Step 3)** Allow fighter to move with elementary input command under update() method(not yet working with input handler, this step would be significantly refactored once input handler is hooked up)

if(keyDown(KeyEvent.*VK\_LEFT*){  
 tree.move(pen, -1,0); //pen is a graphics2D object

} //and similarly for all four directions.

**Complex Use Case:** Create a fighter made of a torso sprite, head, upper arm, lower arm, thigh, legs, while holding a weapon; This fighter can perform multiple animated attack combo moves.

**Step 1)** Create a LimbNode for every body part, specifying distance from respective parents

LimbNode torso = new LimbNode(“torso”, imgTorso, this.getWidth()/2, this.getHeight()/2); //parent node doesn’t take dx, dy, angle theta

LimbNode LeftArm = new LimbNode(“LeftArm”, torso, imgLeftArm, -15, -, 45);

//LeftArm is a child node, it has a separate, more detailed constructor that takes a parent, dx from parent, dy from parent, rotated angle from initial position

LimbNode LowerLeftArm = new LimbNode(“lowerLeftArm”, LeftArm, imgLLeftArm, 0, LeftArm.getHeight()/2, 45); //repeat for all limbs

**Step 2)** Add LimbNodes in appropriate hierarchy

LeftArm.add(LowerLeftArm);

Torso.add(LeftArm);

**Step 3)** Add to a BodyTree

BodyTree tree = new BodyTree(Torso);

**Step 4)** Create Animation and set animation active with user input

Animation punchingMotion = new PunchAnimation();//all animation objects extend animation abstract superclass

… if(keyPressed(KeyEvent.*VK\_SPACE*){

punchingMotion.activateAnimation();//sets isActive Boolean to true

}

**Step 5)** Animate the corresponding limbs when receiving input command, under update() method (Current Version, subject to change)

If(punchingMotion.isActive() == true){  
 pounchingMotion.animate(LeftArm, LowerLeftArm);

}

**-----------------------------PART II DESIGN DESCRIPTION ---------------------------------**

**Overview:** The main feature of this design stems from the decision to treat each body part of a fighter as a separate sprite, associated with the torso through a tree structure/composite pattern with the torso as the root and all other body parts as children or children of children, etc.

**Main Classes and Description:**

**-BodyTree/Fighter**

-BodyTree is the pointer to the root of the fighter tree

-It constructs a HashMap<String, LimbNode> based on the content of the root node, allows fast access to a particular node in a BodyTree without having to traverse the tree

-**LimbNode (extends Sprite)**

-LimbNode stores all the information about a body part: own image, its parent, its distance from parent, its angle of rotation, its own children LimbNodes; and if the developer wants, one can also store information such as damage taken, damage multiplier, etc.

-render() method renders the fighter sprite tree recursively. It takes x coordinate, y coordinate, and a base angle of rotation. It is also responsible for moving sprites and rotating sprites. Rotation of sprites is calculated through rotation matrix.

\begin{bmatrix}
x' \\
y' \\
\end{bmatrix} = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta \\
\end{bmatrix}\begin{bmatrix}
x \\
y \\
\end{bmatrix}

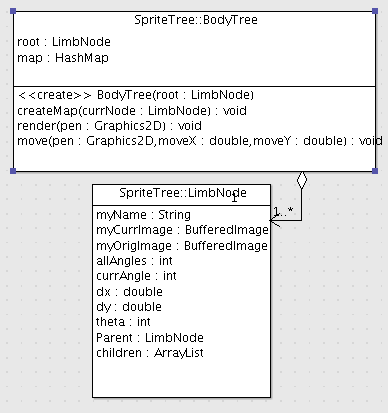
In code:

Double dx =

Math.*cos*(Math.*toRadians*(baseTheta)) \* **this**.dx - Math.*sin*(Math.*toRadians*(baseTheta)) \* **this**.dy;

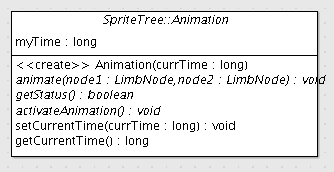
Double dy =

Math.*sin*(Math.*toRadians*(baseTheta)) \* **this**.dx + Math.*cos*(Math.*toRadians*(baseTheta)) \* **this**.dy;



-**Animation** and **Motion** objects (to be implemented)

-Animation is an abstract object, its subclasses calculate animation frames based on the number of updates performed. Each Animation object keeps a boolean switch that determines whether the animation is activated/in progress, or deactivated/ended. And each has its own animate() method that determines how many limbs to take, and what to do with these limbs.



-Motion object is created for the developers using this framework to define combo moves through configuration files, instead of having to hard code every combo movement. In this design, animation is defined in terms of limb, angle, and the amount of time/update calls it takes to get there. A Motion object simply constructs itself form a configuration file by taking: 1) the name of limb that is about to perform an animation, 2) the amount of time it takes to complete the animation, 3) the angle which the limb will rotate about its joint. Once the configuration file is read, we will have a ArrayList<Motion> to work with, as shown by pseudocode below:

ArrayList<Motion> mySequence;

Motion myCurrentMotion = mySequence.getItemAtIndex(0);

Boolean activate; …

If(mySequence.length() ==0){

Activate = false; //finished processing all motion objects, the animation ends

}

else{ //if the animation is active

if(Map.get(myCurrentMotion.getLimbName()).getAngle() != expectedAngle){

int diff = expectedAngle – Map.get(name).getAngle();

map.get(myCurrentMotion.getLimbName()).rotate(diff / myCurrentMotion.getTime());

//rotate the limb toward the specified angle in specified time

}

else{ //if finished rotation, then ready to process the next motion

mySequence.remove(myCurrnetMotion);

myCurrentMotion = mySequence.getItemAtIndex(0);

//get the next motion in sequence

}…