**Design Doc Milestone III**

**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 INTRODUCITON----------------------------------**

**i) Coordinating movement of fighter body parts**

**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);

}

**ii) Define animation for any arbitrary limb of a fighter**

**Simple Use Case:** Simple Animation creation, performed on one node, only perform once when called:

-Assume that we already have a FighterBody :  
 LimbNode torso = **new** LimbNode("torso",imgT, 50, 50);

myTree = **new** FighterBody(torso); (\*\*formerly BodyTree)

**Step 1)** Define Motion object for a particular limb, given the name of the limb sprite to be rotated, the angle of rotation, the FighterBody on which the limb is attached, and the amount of time in elapsed time it takes the limb to complete the animation:

Motion m1 = **new** Motion("torso", -800, myTree, 400);

**Step 2)** Create a HashMap that maps the start time of the Motion in the overall timeline and the Motion performed:

HashMap<Long, Motion> sequence = **new** HashMap<Long, Motion>();

Sequence.put((long) 0, m1);

**Step 3)** Create your Animation Object, that takes a HashMap<Long, Motion> and the FighterTree it’s animating:

**this**.animation = **new** Animation(sequence, myTree);

**Step 4)** Activate the animation upon receiving Input Handler command:

**this**.animation.activateAnimation();

**Step 5)** Update the animation each update call:

**if**(**this**.animation.getStatus()==**true**){

**this**.animation.update(elapsedTime);

}

**Complex Use Case:** make a stick figure dance the can-can, the stick figure has a torso, arms, legs, lower legs, the character performs the action repeatedly until given command to stop

-Assume we have built a stick figure as following:

RightLeg.addChild(LRightLeg);

LeftLeg.addChild(LLeftLeg);

torso.addChild(LeftLeg);

torso.addChild(RightLeg);

torso.addChild(RightArm);

torso.addChild(LeftArm);

torso.addChild(head);

myTree = **new** FighterBody(torso);

**Step 1)** Define Motions:

Motion m1 = **new** Motion("RightLeg", -800, myTree, 400);

Motion m2 = **new** Motion("LRightLeg", 900, myTree, 400);

//…repeat for the rest

**Step 2)** Create timeline map and put motions into the map

HashMap<Long, Motion> sequence = **new** HashMap<Long, Motion>();

sequence.put((**long**) 1003, m1);

sequence.put((**long**) 1004, m2);

//…repeat for the rest

**Step 3)** Create Animation and set to repeat

**this**.animation = **new** Animation(sequence, myTree);

**this**.animation.setRepeat(**true**);

**Step 4) +5)** To activate animation, repeat step 4) and 5) from previous example

**----------------------------------PART II DESIGN DECISION----------------------------------**

**Overview:**

**I) Coordinate Sprite Parts**: 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.

**II)Defining Animation:** Each Animation object defines a particular movement a fighter sprite might perform,(i.e. walking, punching, kicking, etc.) and is consisted of 1 or more Motion objects.

**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.

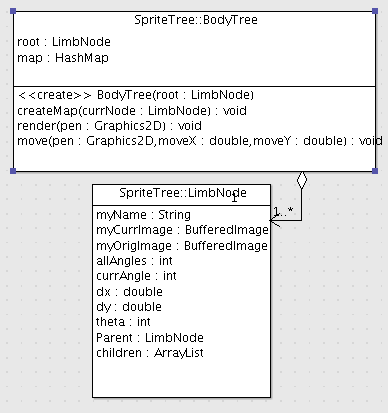
-render() method renders the fighter sprite tree recursively. It takes x coordinate, y coordinate, and a base angle of rotation. It is also responsible for moving sprites and rotating sprites. Rotation of sprites is calculated through rotation matrix.

\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;



-**Motion**

-Motion defines how much a limb should move, by a specified angle in a specified time, in milliseconds.

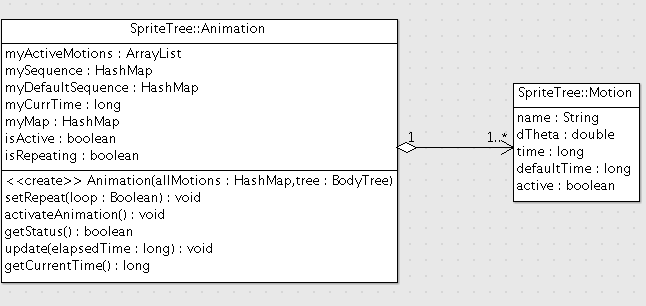
-It takes the expected angle and the current angle of the limb node (to be moved) and computes the angle it should rotate by every millisecond.

-When update() is called, the field that keeps track of the amount of time the Motion should last decrement by elapsed time.

**-Animation**

-Animation is defined by the Motions associated with it and the timing of the Motions. It holds a HashMap that maps the timing of a motion to the appropriate motion defined by the user.

The aggregation relationship is illustrated by the following UML:



-Animation’s current time starts with 0, adds elapsed time to its current time upon each update. This time forms the timeline for the Motions.

-For each entry in the HashMap, if the timing of the Motion falls within current time, the Motion is added to an arrayList of active motions, ready to be performed.

This design is illustrated as follows:

