**Classes/Structures**

**Point2D:** A 2D point consisting of 2 coordinates. On top of this, should include some methods and operators for handling interactions of points. (Can use a Vector2 class instead)

* *float x* 
  + The x coordinate of the point
* *float y*
  + The y coordinate of the point
* *static float findSlope(point2D point1, point2)*
  + Should find the slope between point1 and point2

**Shape:** A class that handles shapes for collision detection. Will store an array of Point2Ds, then use SAT to actually check collision between other shapes. Can potentially have specific shapes as sub classes of this one (Rectangle, triangle, etc.). If we have a circle class, we may have to have special checks in place, as a circle would not work as an array of vertices.

* Point2D[] vertices
  + Array of the vertices as 2D coordinates for this shape
* Shape projection(axis)
  + Returns the projection of the calling shape onto the given axis. Will be used In collision detection. Should return an array of Point2Ds or a shape. Basically, this should flatten the shape to the projection axis.
    - Should find where the line perpendicular to axis passing through each vertex intersects the axis. Can find the equation of each line for each vertex by taking (vertex.x + axis.slope\*vertex.y – axis.slope\*axis.y\_int) / (axis.slope\*\*2 + 1) for all vertices.
* static bool checkCollision(shape1, shape2)
  + Checks if shape1 and shape2 are colliding using SAT. The general idea:
    - Find a set of axes perpendicular to each edge of each shape. Essentially, use line and Point2D helper functions to create a line with a slope and y\_int perpendicular to an edge, and then store it in an array.
    - For each axis, take the projection of shape1 and shape2. Check if the projections then overlap. If they don’t, bail out with a false. Consider creating function to convert flattened shapes to a LineSegment.
    - After taking all the projections, if we didn’t return early, return a true.

**Line:** A 2D line with a slope and y-intercept.

**LineSegment : Line:** A class that extends Line. Adds in two Point2D end points.

**Vector2\_generic:** Generic 2D vector class used for vector operations. Made generic to support Vector2Int/ Vector2Float/ Vector2Double, etc. Basically extended Point class, or even sub class on Point.

**StateMachine/State :** StateMachine object which holds all State objects within itself. This state machine object calls the states update functions for that frame and handles all transition cases for the current state.

**Alternative State Machine :** State objects hold references to their next states and only the current state is kept track of.

**AnimatorController :** Class that holds animations and can control its animations list using a small state machine. (i.e. an enemy class would have one where we would supply walking/attacking/death animatons and the controller would play them correctly)

**Animation :** Simple structure to hold a series of sprites with a few settings about the animation itself. Looping? Duration? Finish current animation before transition?

**Notes/ things possibly needed:**

* Animation class that handles an array of sprites and cycles through them. Couldn’t find a class already in the engine, but I may have missed it.
* There seems to be a polygon collision checker source file included in the olcGameEngine repo.
* Something to handle controls. Again, might already be in the engine.

\*I don’t believe there is anything already there for sprite animations, but I’ve experimented with that and was fairly easy to implement.

\*The author of the engine has done some convex polygon collision examples so we may be able to “borrow” that.

\*The input system is very easy to use and make an input manager ( GetKey(olc::A).bPressed → returns true if the A key is pressed, but not held)

* InputSchemes/KeyBinding Layouts : allow players to set and rebind keys from within the game. This layout should be encapsulated for serialization.
* Simple State Machine : used to control the different “game states” and their transitions