***Main algorithm for simulating the game***

This algorithm takes an input game which was shown on the previous graphics frame, and gives an output game, that should be shown on the next graphics Frame.

**The Key idea of the algorithm, which I call TimePoint algorithm, is to find the EARLIEST collision that will end the game (if any) and advance the game up to that point (for all entities). If there is no collision about to happen (over 1 time unit max), then it will advance the game to the next state based on a common time.**

**The main algorithm proceeds in 6 phases (in XoMotion.hs):**

**Phase 1**: Get minimum possible player trajectory. (player movement in either 1 time unit, or x time units as explained in algorithm below)

**Phase 2**: Get an **independent** list of trajectories for each non-player moving entity, assuming that it will move like it is the only moving entity in the game (i.e. no collisions). (foreach entity get [(time, point, direction)]). I call these timepoints.

**Phase 3**: For each moving entity, determine **\*earliest\*** collision of trajectories from phase 2 with player trajectory from phase 1. (foreach non-player moving entity (collHappened, colPoint, colTime)

**Phase 4**: For each ball, compute **\*earliest\*** possible collision with pre-existing polyLine trace. Also compute collision w/ newly generated polyline segment portion. Then compute if player itself has collided with polyline.

**Phase 5**: If any of phase 3 and 4 have collhappened then life will end, else game will continue. If collhappened, then calculate the earliest collision based on various colTime. (so now we know if life ends or not, and if it ends, at what time.)

**Phase 6**: Advance all the entities over time determined by phase 5. End game or life if needed, else send final state for rendering to HOpenGL.

**Disclaimer:** Strictly speaking, any collision will end that life (or the game) for Xonix, so strictly speaking, one could argue that the TimePoint algorithm isn’t really needed for simulation (after all, who cares if all the other entities are in their right place when Xonix dies?). However, the basic idea behind TimePoints can be applied to other games/problems that don’t terminate on a collision, so I think it’s a useful thing to have.

Also, for a pictorial description of timepoints, refer to the end of this document.

**Phase 1:** This phase only evaluates the potential player trajectory.

* First we evaluate the trajectory of the player. i.e. where it moves from and to in 1 time unit (given player speed in pixels/time-unit).
* The next step is only meant to find out whether a new polyLine has been started (player enter polygon), or the polyLine eliminated (player exits polygon) due to the above player trajectory.
* For the player trajectory, we find the first intersection point of the trajectory with any BB edge.
* If it does not exist, then the player is not starting or ending a polyLine (it may be continuing one though, which is ok).
* If it exists, then the player trajectory is reduced to this point, and the time reduced to this time.
* This step makes the problem more manageable later, so that we don't have to consider the case of multiple polyLines resulting from multiple cuts of multiple polygons due to a very fast player speed.

**Output of Phase 1: At the end of this phase, we have a player trajectory from start point to finish point. It either finishes at the start/end of a polyLine trace (if this event exists prior to 1 time unit), or at the end of 1 time unit. Time taken = x time units.**

**Phase 2:** In this phase,

* Each remaining moving entity (balls, monster) is evaluated, **as if it is the only entity in the game.** i.e. we take the initial position, direction, speed of an entity, and x time unit travel, and evaluate a list of trajectories (with time per trajectory) it will take. We need a list of trajectories, because of potential bouncing off of the walls for balls (multiple bounces possible if the ball speed is really high w/ game difficulty, or at corner).

XoGame -> XoEntity -> XoTime -> [(XoTime, XoLine, XoDirection)]

**Output of phase 2: At the end of phase 2, we have for each moving entity in the game, a list of timePoints, which tell us at what point the entity was, at what time, in what direction (for each change of direction).**

**Phase 3:** Now we know all the potential trajectories of all the entities in 1 (or x) time unit from phase 1 and 2. In phase 3,

* For each non-player entity (balls, monsters), given their trajectory lists, we compute if any trajectory in that list collides with the player trajectory. If yes, then we have a \*potential\* collision between player and entity.
* The potential collision is a real collision, only if both the entity and the player arrive at the collision point at the \*same time\*.
* Note that for a ball-player collision, the ball can arrive at the potential collision point later, and encounter a polyline, which should end the game. This is computed in phase 4. For the sake of this phase, we only consider ball arriving at \*same\* time and not later.
* Also note that, again, for a ball-player collision, the game may end if the ball hits the prior polyLine constucted (from earlier simulation) long before this player trajectory. that is ok, because it is handled in phase 4 as well.
* For each entity, given a (possible) list of collisions of that entity with the player, we sort and take the **earliest (first) real collision**.
* Thus for phase 3, each entity will return a triple of (collisionHappened::Bool, endPoint::XoPoint3D, endTime::XoTime)
* EndPoint is defined as:
* If collisionHappened is true the endPoint is collisionPoint
* Else if no collision, it is the endPoint of the actual trajectory of the entity.
* endTime is defined as:
* If collisionHappened is true, it is the collisionTime.
* else it is the end time of the trajectory (should be 1 or x).

determineEarliestDynamicCollision ::

XoGame -> XoTime -> XoSize -> (XoPoint3D, XoPoint3D) -> TimePoints -> Maybe (XoTime, XoPoint3D)

Note that the collision detection tests are checking for \*point-point\* collisions. The balls, monsters and players are actually entities having a width, so one could argue that point-point collision tests are not only inaccurate, but may result in cases where a real collision could not be flagged. (i.e. where ball-ball would collide, but point-point would not collide). This is very true, and is a drawback of my simple line-line collision detection. To be more accurate, you want polygon-polygon collision detection using separating axis theorem. I deem it to be lower ROI, since line-line is mostly good enough.

**Output of Phase 3: At the end of phase 3, we have for each non-player entity, a triple indicating whether that entity collided with the player trajectory, and if so when and where.**

**Phase 4:** This phase is meant to find out

1. whether a ball intersects an older part of the polyLine (which wasn't constructed in \*this\* player trajectory). This is static collision detection (static lines w/ moving body).
2. Whether a ball intersects the newer part of the polyline (from old player center to new player center). This is dynamic collision detection (dynamic body w/ dynamic trajectory).
3. Whether the player itself intersects an older polyline segment. This is static collision detection again (dynamic body w/ static lines).

* For each entity, we evaluate the intersection of the entity trajectory with all the polyLine edges as noted above. The closest such intersection point is the first encounter of the entity (ball or player) with the polyline. This is recorded in the same type of triple as phase 2.

(Bool, XoPoint3D, XoTime)

**Output of phase 4: At the end of phase 4, we have for each ball, a triple indicating whether the ball collided with a pre-existing polyline edge, and if so when and where.**

**Phase 5:** This is a very important phase, because it tells us exactly how much \*time\* to advance the game by. So far we know

* The trajectories of all the moving entities in the game, and the times needed for these trajectories.
* The earliest collision point between each non-player moving entity and the player.
* The earliest collisoin point between each ball (or player) and polyLine trace.

Now we simply check if any collHappened (from phases 3 and 4) is true:

* If it is true then:
* For all the triples with collHappened = true, sort all the triples in ascending order of the collTime. Then take the head of the list (minimum collision time). This is where the game will end, because of some collision. We note the time the game will end as time y.
* If it is false then there was no collision, and the game can proceed up to the end of either time unit 1 or time unit x (as determined by phase 1). We note this time as time y.

**Output of phase 5: An indication (bool) whether the game ends or not, and if yes then at what time (time y). If the game does not end, then it tells us to advance each entity over time y.**

**Phase 6:** This phase finally puts all the entities at their final places, and/or also ends the game if any collision happened. Given:

* A list of timed trajectories from phase 1 and 2, and
* A value for the actual time to advance the game to from phase 4,
* We can compute using the above two, the actual positions to advance each entity to.

Also we do some simple scoreboard calculations in this stage.

**Output of phase 6: This phase ends the game (or life) if needed. Either way, whether the game (or life) ends or not, it puts all the entities in their final places, to be ready for rendering on the screen by HOpenGL.**

**What is a timepoint?**

TP0

TP2

TP1

Consider an entity (ball) moving as shown above. The list of timepoints (TP0, TP1, TP2) gives us the exact location of the entity at any point of time. (For any entity moving with constant velocity or constant linear acceleration).

A timepoint is defined as a triple of (time, starting point, starting direction). E.g.

TP0 = (T0, P0@(px0, py0, pz0), D0@(dx0, dy0, dz0))

So at time T0, the entity was at P0, moving in direction D0. (Constant velocity assumed for Xonix entities). Using this information, we can figure out where the entity was at any time. This helps in collision detection. TP1 and TP2 are defined similarly.

**Some notes on collision detection.**

Note that the game uses an a-priori (continuous) vs. a-posteriori (discrete) collision detection algorithm. The advantage of the former is that we find the exact point of intersection *before* the collision actually happens. This is done mathematically using line-line intersection, so it is quite exact. Also, with very fast speeds, the a-posteriori algorithm may miss the collision completely if that step-size advances the colliding entities past (i.e. through) each other.

Furthermore, for dynamicbody-dynamicbody collision detection, I also make sure that not only do they collide, but they arrive at the collision point at the same time. This will faithfully simulate the collisions.

However, the one shortcut that I have taken is to do a line-line intersection for collision detection (instead of drawing out the polygons corresponding to the movement and doing polygon-polygon collision detection). The former is simpler, but less accurate, since the latter will catch cases that the former may not. If you feel adventurous, feel free to re-vamp the collision detection in the game (using for example the Separating-Axis-Theorem) for accurate collision detection.

**Some known issues with the game:**

**Ball freeze/stuck:**

* Sometimes the ball freezes or gets stuck on in a very narrow section. This causes the game to hang.
* I strongly suspect that the problem is in the ball timepoint generation code, which keeps trying to generate timepoints until the frameTime is all used up. Obviously if the section is narrow such that the ball gets stuck (old center == new center), the timepoint does not advance the ball, and hence takes 0 time. This keeps the recursion going, and it generates infinite timepoints, thus hanging the game there.
* **A simple fix** would be to clamp it at say (take 10) or so, and rely on lazy evaluation to make the process finite. Of course you have to ensure that the timepoint still records the entire frametime as being consumed (even though there is no movement).

**Too narrow edge to be cut:**

* If the polyline (on entrance or exit into a BB) encounters an edge of width < polyWidth, the subsequently generated points for the two polygons will obviously be wrong. Currently I punt on this scenario with an error message.
* **A possible fix** might be to perturb the entry or exit position of the ball, so that it avoids the narrow edge. Perturbation on exit is probably more complex, since you will likely have to perturb the prior point on the polyTrace as well. I haven’t really tried this fix, but if you are adventurous, you may try it!

**Cutting on a corner:**

* If the ball enters the BB on a corner, then the intersection algorithm will find 2 edges for the entry point (the 2 edges forming the corner).
* Also, on exit, the polyline will cut the BB into 2, however they will have edges that are not parallel to x/y as a result of the cut (since the cutpoints are off from the corner by polyWidth/2). This will mess up future cuts, since the code (rightly) expects the edges to be x/y parallel.
* **A possible fix** would be to perturb the entry point by polyWidth/2 in the correct direction, so that on exit, the points are in line with the requirements.

**Why are the monsters moving so weirdly?**

* That’s because they need a better routing algorithm! Currently I just draw a straight line from the monster to Xonix, and try to make them move along that line. This is very naïve, and especially at the start of the game they look very lethargic.
* With some effort, we can design a smarter routing algorithm for the monsters. This will be a future modification!