Write-up for Initial C++ Class Development and Module Ideas - Group 13

Coordination With Other Groups:

- Group 11 is using the Point class from Team 13 for their WorldPath class which keeps track of the location history and future trajectories of agents.
- Potential for coordination with group 20 for sharing data about collisions between objects.

CLASSES

`Point` A point in 2D geometric space, with an X and Y coordinate. This class should have many different functionalities for working with pairs of points, including standard mathematical operations, rotations, scaling, dot products, etc., depending on what will be useful for projects in your company.

The point class represents a point or 2-dimensional vector and provides arithmetical operations (vector addition and scalar multiplication), geometric functions (dot product, cross product, lengths, normalization), and conversions between types of C++ point objects.

`Circle` - A simple geometric shape that tracks its position and radius and can identify overlaps with other Circles or if a given Point lies inside of the circle. You should make sure that it has a set of functionality that will be useful for projects in your company.

Create a simple circle with a center point and non-negative radius. Provides containment, overlap test, and intersection helpers, movement/scale operations.

`Box` - Similar to `Circle`, this will define a simple geometric rectangle that tracks its position and dimensions and can identify overlaps with other shapes or if a given Point lies inside of it. You should make sure that it has a set of functionality that will be useful for projects in your company.

`Surface` - An area that tracks a set of shapes, identifying all overlaps; particularly useful as the basis for a simple physics model. Make sure it can handle shapes moving and detecting overlaps as soon as they occur. For a speedup, you can either use sectors (where a circle can only overlap with circles in the same or neighboring sectors) or a more sophisticated quadtree.

`Scheduler` - Schedules a set of processes based on a "priority" measure and returns the ID of which one should go next. The priority of a process should be proportional to how often it is scheduled. This class can be built to be probabilistic or evenly integrated.

POINT:

Class Description: Has been done (as in 1.1)

Similar Classes in C++:

1. std::pair: in the <utility> header for reference as a two-value container but they don't have any built-in math semantics.
2. std::complex: in the <complex> header gives us more mathematical power. It supports arithmetic on a pair of numbers.
3. std::tuple: is a variadic template that can hold more values than std::pair.

Key Functions:

1. Constructors: default constructors and parameterized constructors
2. Operators (for arithmetics): operator+, operator*, operator==, etc.
3. Geometry: dot and cross product, magnitude, normalize(return unit vector),
4. Transforms: rotate, scale

Error Conditions:

1. Index out of bounds: if we provide access through operator[] and index not in Point object
2. Division by zero will trigger an assertion, since this violates a precondition by the caller
3. Error (2) above leads us into the fact that you cannot normalize a zero vector because length will be zero (0)

Expected Challenges:

1. Floating Point Precision: Since we are doing math-heavy simulation, strictly comparing two floating point items can be dangerous due to precision errors. A robust epsilon-comparison system has to be developed to catch these cases.
2. Normalization edge cases: handling near-zero length vectors safely.

SCHEDULER:

Class Description: The scheduler is going to be a utility that manages what happens next in the world/simulation based on priority. It will ensure that high importance agents, processes or tasks are processed at the correct intervals relative to others in the world. e

Similar classes in C++:there are the std::priority_queue , std::discrete_distribution , std::queue

Key Functions: The Scheduler class is going to have the following functions

1. ProcessID object?
2. Void AddProcess(int id, double weight) : This will register a process with a specific weight or priority
3. Int GetNext() : This will return the ID of the process that will be executing next based on its priority
4. Void UpdatePriority(int id, double newPriority): This will adjust the priority of a particular ID
5. Void ClearScheduler() : This will clear the scheduling counters.
6. bool Empty() - This will indicate whether any processes or task remain int eh scheduler
7. Int Size() - This will return the number of tracked processes

Error Conditions:

1. Negative Priority Weight: There need to be a mechanism to check adding a negative weights
2. Requesting from an empty scheduler.
3. If there are competing priorities there need to be a tight breaker.

Expected Challenges:

1. One challenging is balancing the scheduler so that there is a fair chance such that low-priority tasks doesn't starve completely
2. Choosing between deterministic and probabilistic scheduling. There need to be a balance between

Coordination With Other Groups

1. For collaboration, the scheduler class may be helpful to Group 20 which is the Data Analytics group for timing information

Box:

Class Description: The Box class represents a simple axis-aligned rectangular shape in 2D space, defined by a position (typically the lower-left or center Point) and dimensions (width and height). It is designed to support spatial queries such as point containment and overlap detection with other shapes. The Box class will be useful for collision detection, region queries, and bounding areas in simulations or game-like environments.

Similar Classes in C++: std::pair / std::tuple, std::array<double, 4>, <algorithm>

Key Functions:

The Box class is going to have the following functions:

- Constructors (default + parameterized with position and size)
- Point GetPosition() (or GetCenter(), depending on convention)
- double GetWidth()
- double GetHeight()
- void SetPosition(Point p)
- void SetSize(double w, double h)
- bool Contains(Point p) – checks if a point lies inside the box
- bool Overlaps(const Box& other) – checks if two boxes intersect
- void Translate(Point delta) – moves the box
- void Scale(double factor) – scales width and height

Error Conditions:

- Negative width or height (programmer error → assert)
- Negative scale factor (programmer error → assert)
- Invalid dimensions (zero or extremely small sizes may cause edge cases)
- Overlap tests simply return false when no intersection occurs

Expected Challenges:

- Choosing a consistent definition for position (center vs corner)
- Handling edge-touching cases (do touching boxes count as overlapping?)
- Maintaining compatibility with Circle and Point APIs
- Floating-point precision near boundaries

Coordination With Other Groups:
The Box class will coordinate with:

- The Circle class for consistent Contains() and Overlaps() interfaces
- The Surface class for collision detection and spatial indexing
- Any simulation or world modules that rely on rectangular regions

Surface:

Class Description: Manages a collection of shapes (e.g. circles, boxes, and other custom objects) and detects overlaps between shapes. Tracks moving shapes and reports when they occur. The key use for this class is basic collision detection for physics, triggering logic, spatial queries, etc.

Similar Classes in C++: std::unordered_map, std::vector, std::priority_queue

Key Functions:

1. ShapeID addShape(const ShapeRecord& rec); Add Shape to data structure.
2. bool removeShape(ShapeID id); Remove Shape from data structure.
3. bool updateShape(ShapeID id, const ShapeRecord& newRec); Update the geometry of the shape.
4. bool translateShape(ShapeID id, Point delta);
5. std::vector<std::pair<ShapeID, ShapeID>> detectAllOverlaps() Detect overlaps between shapes.
6. std::vector<ShapeID> queryRegion(const AABB& region) const; Get shapes from given region.
7. std::vector<ShapeID> queryRadius(Point center, double radius) const; Get shapes from provided radius.

Error Conditions:

1. addShape being used with an unsupported shape results in an assert or exception.
2. If there are too many shapes taking up too much memory, throw std::bad_alloc or return a failure.
3. Invalid shape ID results in returning false or std::out_of_range being thrown.

Expected Challenges:

1. Consider using grid sectors or a quadtree to provide a more efficient implementation.
2. Tune cell sizes and object sizes to optimal ratios for better performance.
3. Dynamically update individual objects, avoiding global rebuilds for enhanced efficiency.
4. Consider collision filtering to avoid unnecessary testing.

Circle:

Class Description: The Circle class represents a simple 2D geometric shape defined by a center Point and a non-negative radius. It is intended to support simulations and spatial modeling by providing utilities to test whether points lie inside the circle, whether two circles overlap, and to allow basic transformations such as movement and scaling. This class will be used for collision detection, containment checks, and general geometry operations in the system.

Similar Classes in C++: std::complex, std::pair/std::tuple

Key Functions:

The Circle class is going to have the following functions:

- Constructors (default + parameterized with center and radius)
- Point GetCenter()
- double GetRadius()
- void SetCenter(Point c)
- void SetRadius(double r)
- bool Contains(Point p) – checks if a point lies inside or on the circle
- bool Overlaps(const Circle& other) – checks if two circles intersect
- double DistanceTo(Point p) – distance from center to point
- void Translate(Point delta) – moves the circle
- void Scale(double factor) – scales the radius

Error Conditions:

- Negative radius (programmer error → assert)
- Negative scale factor (programmer error → assert)
- Floating point edge cases when circles barely touch
- Intersection functions may return empty results when no overlap exists

Expected Challenges:

- Handling floating-point precision, especially for tangency and overlap checks
- Deciding whether touching circles count as overlapping
- Ensuring consistent behavior with the Point class
- Optimizing distance calculations (using squared distances instead of sqrt)

Coordination With Other Groups:

The Circle class will coordinate closely with:

- Group 11, which uses Team 13's Point class for their WorldPath system
- Our own Surface class for collision detection and spatial queries
- The Box class, to maintain consistent interfaces such as Contains() and Overlaps()

VISION FOR THE MAIN MODULE

Depending on the company's decision, possible ideas include a simulation of disease spread, an astronomical simulation, pinball tables, or otherwise some game-like simulation that allows for objects/shapes to overlap.