Rudimentary 2D Physics Engine

Collin Jones | Medhaj Shrestha

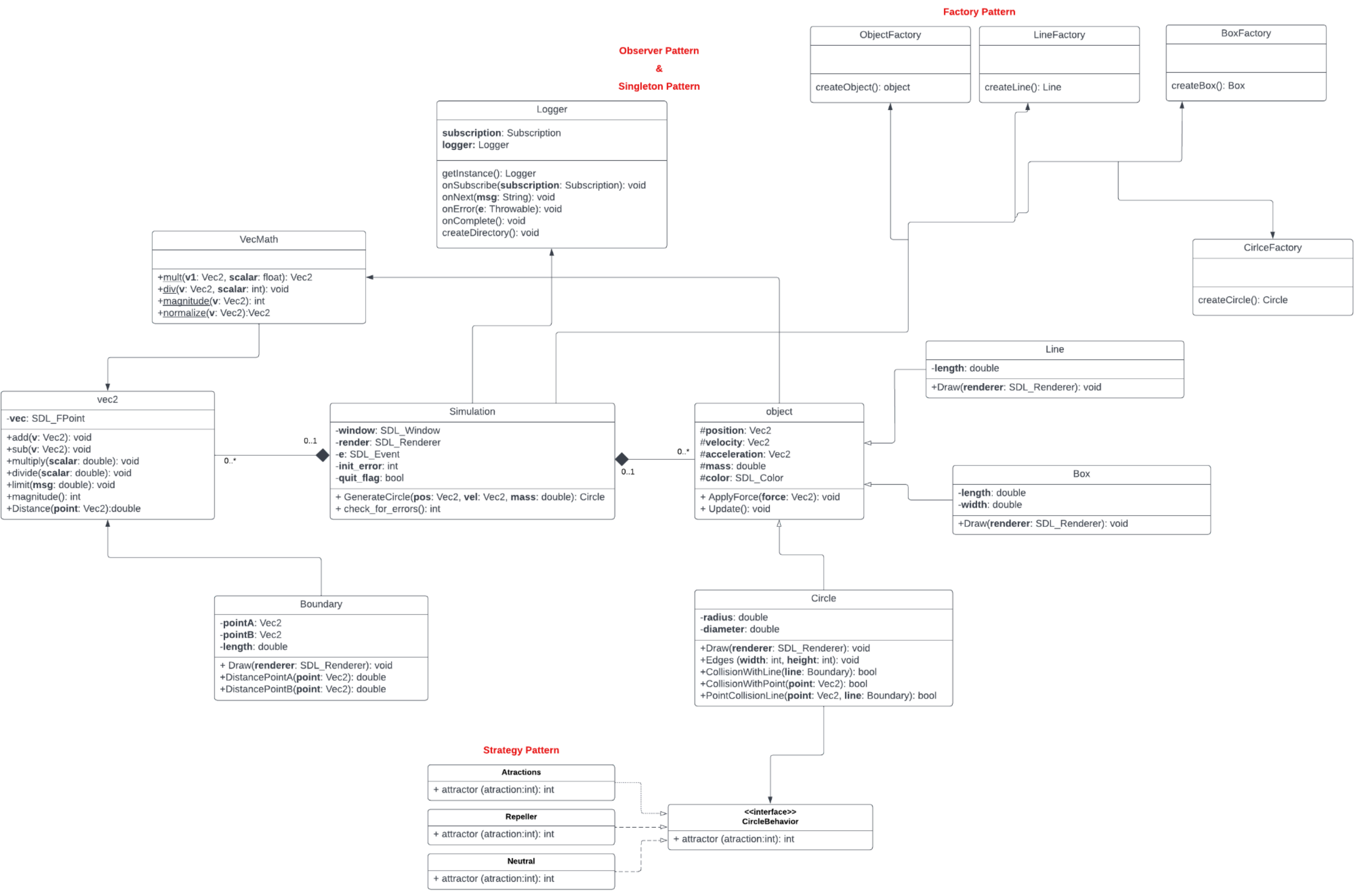
Final State: The final state of our project is one where the user can create an environment of circles, lines, and rectangles. The user can view these shapes that interact with each other through their collisions and continually evolve their environment. No environment has to be the same because there is a GUI for the user to design however they want, like a sandbox game. There are two preset environments available for the user, a solar system and a pachinko machine. A list of the features implemented: can add a circle (repeller, attractor, or neither) anywhere on the screen, can add a line anywhere on the screen, can add a rectangle anywhere on the screen, can view a solar system environment, can view a pachinko machine environment, can turn gravity on or off, clear the screen of all objects, write any creation or deletion logic to a results.txt file with their respective timestamps. Some features not implemented: The ability for a user to delete specific objects or change the attraction state of a circle. This was not implemented as it would take new logic to get the location of the mouse button click and if an object was present there. Another feature not implemented was being able to give the initial velocity of added circles rather than have a default of 0,0. We did not implement this as we could not figure out how to pass the double values into the GUI window and create a circle like that. Another feature we did not add was the ability to pause the simulation and change the environment when paused. We did not add this because when the simulation would be paused, the renderer would not have been able to render the object as it also would have been paused. Overall, we believe we could have added these features given more time. A big change from project 6 and our initial plan was the use of nanogui. Nanogui was too old and was not kept up so it was not usable. We had to rather create multiple SDL windows (the same type as our main simulation screen) and create our own buttons. We also had to figure out our own collision logic between the buttons and the mouse to determine if a button was pressed and on which window. Another change from project 6 was not having the state pattern for changing a circle's attraction logic because we could not get that new logic to work. Overall, our project has stayed fairly consistent since the plan in project 5.

Final Class Diagram:

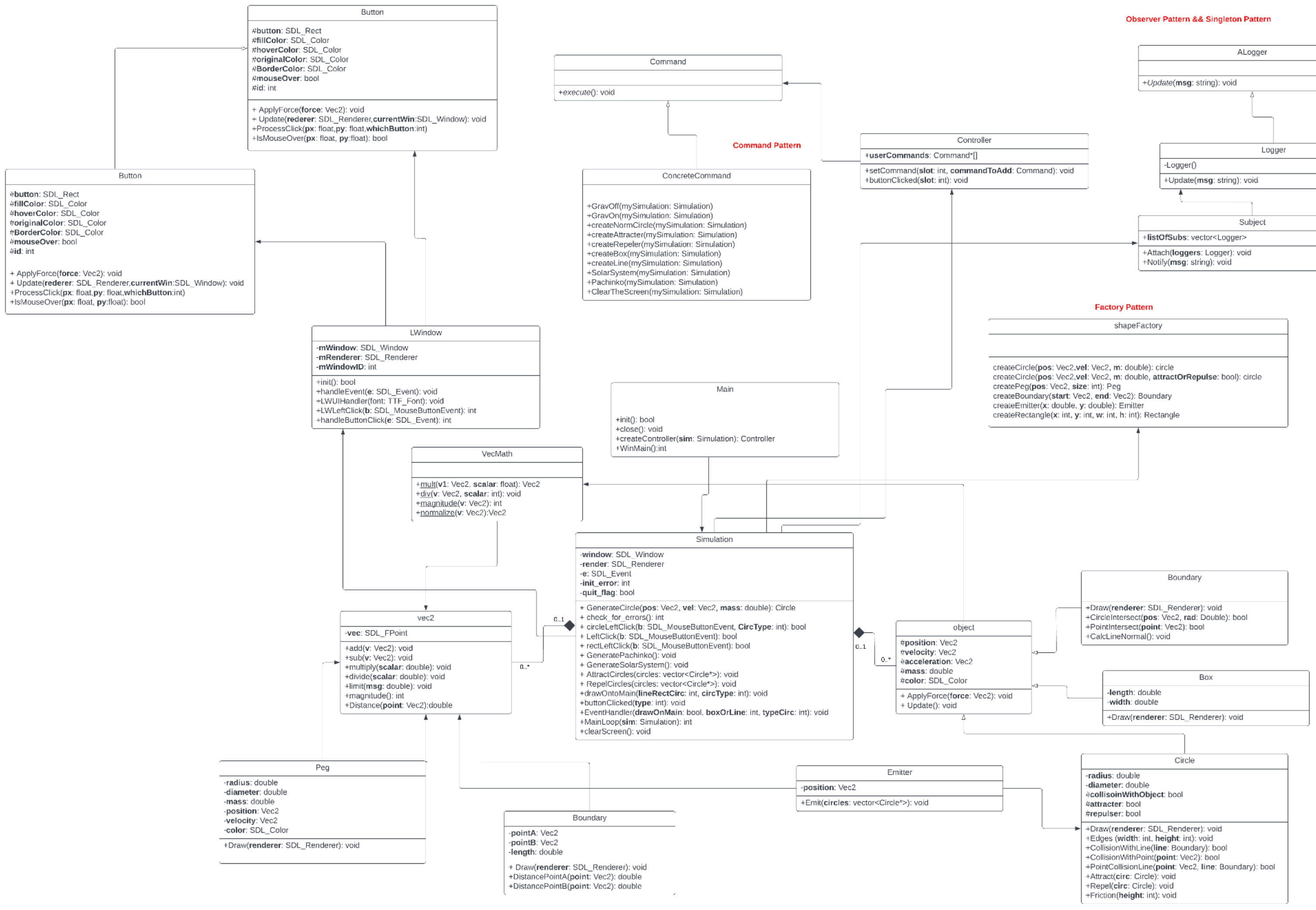
Links to the images if they can not be seen well in pdf:
Project 5 class diagram: **UML Class Diagram & Pattern Use**

Project 7 class diagram: Final UML Class Diagram

Project 5:


Factory Pattern
ObjectFactory
createObject(): object
LineFactory
createLine(): Line
BoxFactory
createBox(): Box
CirlceFactory
createCircle(): Circle
Observer Pattern
&
Singleton Pattern
Logger
subscription: Subscription
logger: Logger
getInstance(): Logger
onSubscribe(subscription: Subscription): void
onNext(msg: String): void
onError(e: Throwable): void
onComplete(): void
createDirectory(): void
VecMath
+mult(v1: Vec2, scalar: float): Vec2
+div(v: Vec2, scalar: int): void
+magnitude(v: Vec2): int
+normalize(v: Vec2):Vec2
Line
-length: double
+Draw(renderer: SDL_Renderer): void
vec2
-vec: SDL_FPoint
+add(v: Vec2): void
+sub(v: Vec2): void
+multiply(scalar: double): void
+divide(scalar: double): void
+limit(msg: double): void
+magnitude(): int
+Distance(point: Vec2):double
0..*
0..1
Simulation
-window: SDL_Window
-render: SDL_Renderer
-e: SDL_Event
-init_error: int
-quit_flag: bool
+ GenerateCircle(pos: Vec2, vel: Vec2, mass: double): Circle
+ check_for_errors(): int
0..1
0..*
object
#position: Vec2
#velocity: Vec2
#acceleration: Vec2
#mass: double
#color: SDL_Color
+ ApplyForce(force: Vec2): void
+ Update(): void
Box
-length: double
-width: double
+Draw(renderer: SDL_Renderer): void
Boundary
-pointA: Vec2
-pointB: Vec2
-length: double
+ Draw(renderer: SDL_Renderer): void
+DistancePointA(point: Vec2): double
+DistancePointB(point: Vec2): double
Circle
-radius: double
-diameter: double
+Draw(renderer: SDL_Renderer): void
+Edges (width: int, height: int): void
+CollisionWithLine(line: Boundary): bool
+CollisionWithPoint(point: Vec2): bool
+PointCollisionLine(point: Vec2, line: Boundary): bool
Strategy Pattern
Atractions
+ attractor (atraction:int): int
Repeller
+ attractor (atraction:int): int
Neutral
+ attractor (atraction:int): int
<<interface>>
CircleBehavior
+ attractor (atraction:int): int

Project 6:


Observer Pattern && Singleton Pattern
Command Pattern
Factory Pattern
Button
#button: SDL_Rect
#fillColor: SDL_Color
#hoverColor: SDL_Color
#originalColor: SDL_Color
#BorderColor: SDL_Color
#mouseOver: bool
#id: int
+ ApplyForce(force: Vec2): void
+ Update(rederer: SDL_Renderer,currentWin:SDL_Window): void
+ProcessClick(px: float,py: float,whichButton:int)
+IsMouseOver(px: float, py:float): bool
Button
#button: SDL_Rect
#fillColor: SDL_Color
#hoverColor: SDL_Color
#originalColor: SDL_Color
#BorderColor: SDL_Color
#mouseOver: bool
#id: int
+ ApplyForce(force: Vec2): void
+ Update(rederer: SDL_Renderer,currentWin:SDL_Window): void
+ProcessClick(px: float,py: float,whichButton:int)
+IsMouseOver(px: float, py:float): bool
Command
+execute(): void
ConcreteCommand
+GravOff(mySimulation: Simulation)
+GravOn(mySimulation: Simulation)
+createNormCircle(mySimulation: Simulation)
+createAttracter(mySimulation: Simulation)
+createRepeler(mySimulation: Simulation)
+createBox(mySimulation: Simulation)
+createLine(mySimulation: Simulation)
+SolarSystem(mySimulation: Simulation)
+Pachinko(mySimulation: Simulation)
+ClearTheScreen(mySimulation: Simulation)
Controller
+userCommands: Command*[]
+setCommand(slot: int, commandToAdd: Command): void
+buttonClicked(slot: int): void
ALogger
+Update(msg: string): void
Logger
-Logger()
+Update(msg: string): void
Subject
+listOfSubs: vector<Logger>
+Attach(loggers: Logger): void
+Notify(msg: string): void
shapeFactory
createCircle(pos: Vec2,vel: Vec2, m: double): circle
createCircle(pos: Vec2,vel: Vec2, m: double, attractOrRepulse: bool): circle
createPeg(pos: Vec2, size: int): Peg
createBoundary(start: Vec2, end: Vec2): Boundary
createEmitter(x: double, y: double): Emitter
createRectangle(x: int, y: int, w: int, h: int): Rectangle
LWindow
-mWindow: SDL_Window
-mRenderer: SDL_Renderer
-mWindowID: int
+init(): bool
+handleEvent(e: SDL_Event): void
+LWUIHandler(font: TTF_Font): void
+LWLeftClick(b: SDL_MouseButtonEvent): int
+handleButtonClick(e: SDL_Event): int
Main
+init(): bool
+close(): void
+createController(sim: Simulation): Controller
+WinMain():int
VecMath
+mult(v1: Vec2, scalar: float): Vec2
+div(v: Vec2, scalar: int): void
+magnitude(v: Vec2): int
+normalize(v: Vec2):Vec2
Simulation
-window: SDL_Window
-render: SDL_Renderer
-e: SDL_Event
-init_error: int
-quit_flag: bool
+ GenerateCircle(pos: Vec2, vel: Vec2, mass: double): Circle
+ check_for_errors(): int
+ circleLeftClick(b: SDL_MouseButtonEvent, CircType: int): bool
+ LeftClick(b: SDL_MouseButtonEvent): bool
+ rectLeftClick(b: SDL_MouseButtonEvent): bool
+ GeneratePachinko(): void
+ GenerateSolarSystem(): void
+ AttractCircles(circles: vector<Circle*>): void
+ RepelCircles(circles: vector<Circle*>): void
+drawOntoMain(lineRectCirc: int, circType: int): void
+buttonClicked(type: int): void
+EventHandler(drawOnMain: bool, boxOrLine: int, typeCirc: int): void
+MainLoop(sim: Simulation): int
+clearScreen(): void
0..1
0..*
0..1
0..*
vec2
-vec: SDL_FPoint
+add(v: Vec2): void
+sub(v: Vec2): void
+multiply(scalar: double): void
+divide(scalar: double): void
+limit(msg: double): void
+magnitude(): int
+Distance(point: Vec2):double
object
#position: Vec2
#velocity: Vec2
#acceleration: Vec2
#mass: double
#color: SDL_Color
+ ApplyForce(force: Vec2): void
+ Update(): void
Boundary
+Draw(renderer: SDL_Renderer): void
+CircleIntersect(pos: Vec2, rad: Double): bool
+PointIntersect(point: Vec2): bool
+CalcLineNormal(): void
Box
-length: double
-width: double
+Draw(renderer: SDL_Renderer): void
Circle
-radius: double
-diameter: double
#collisoinWithObject: bool
#attracter: bool
#repulser: bool
+Draw(renderer: SDL_Renderer): void
+Edges (width: int, height: int): void
+CollisionWithLine(line: Boundary): bool
+CollisionWithPoint(point: Vec2): bool
+PointCollisionLine(point: Vec2, line: Boundary): bool
+Attract(circ: Circle): void
+Repel(circ: Circle): void
+Friction(height: int): void
Peg
-radius: double
-diameter: double
-mass: double
-position: Vec2
-velocity: Vec2
-color: SDL_Color
+Draw(renderer: SDL_Renderer): void
Emitter
-position: Vec2
+Emit(circles: vector<Circle*>): void
Boundary
-pointA: Vec2
-pointB: Vec2
-length: double
+ Draw(renderer: SDL_Renderer): void
+DistancePointA(point: Vec2): double
+DistancePointB(point: Vec2): double

Key changes: Had to create LWindow, Button, and ToggleButton classes to handle our gui since nanogui did not work. Create Command classes to use a command pattern for our gui. Created Logger classes to handle observer and singleton patterns so we write to a txt file and the console. Lots of new methods added to simulation to handle the button collision logic and events that happen in every window. Opted away from the strategy pattern as we would not be creating circles with specific algorithms as they all needed access to every algorithm. There was also lots of smaller changes from project 5 and project 7.

Third Party Code: We used the SDL library extensively for our graphics and windows. The LWindow class was taken from here:
https://lazyfoo.net/tutorials/SDL/36_multiple_windows/index.php
The code was adapted but uses its skeletal structure to handle multiple windows. The algorithms for attract, repel, friction, and drag were found online (Wikipedia) and the Nature of Code book, but no code was explicitly copied.

Statement on the OOAD process: A positive element of the design process for this final project was doing all the diagrams before starting on the code (our project 5). This was very helpful in mapping our project and class interactions so there was little confusion when actually coding on what classes we need, who they communicate with, and what they need to do. Another positive element we incorporated was testing (nothing official). We tested continuously for every small part of the project so the whole project did not have a million errors when we combined our code. There was not much to unit test for as our project relied heavily on graphics that created a sandbox like game. Another positive and negative aspect was encapsulation. In c++ the files are hidden from each other unless you include it in the header, and this is good as it helps to prevent every class from knowing about each other. But this does make it impossible to have files reference each other (like what was needed for the command pattern). It took some time but we were able to get around that issue by extrapolating what we needed to the main.cpp file and using abstract classes.