C++ Project Repot

Larry S. – Alam Z.

This project underwent several iterations before reaching its desired state. Initially, I encountered performance issues that simplecpp couldn't resolve, particularly slow update times for the visual buffer. To address this, I decided to utilize the graphics.h library, which not only accelerated the process but also simplified some of the visual code. I kicked off the project by laying out the class structure.A diagram of a computer

Description automatically generated

I've organized my project into three main sections: Vector 2D, Physics Objects, and Interactable.

Vector 2D: This section comprises a straightforward struct designed to store positions, along with handy member functions for operations on these vectors.

Physics Object 2D: Here, I've encapsulated various data pertaining to physical objects. This includes properties like position, velocity, acceleration, mass, and charge, among others. It serves as a comprehensive framework for simulating and managing objects within a 2D physics environment.

Interactable: The Interactable section is focused on providing functionality for user interaction with GUI elements. It's built around a simple struct called Element, which facilitates the handling and manipulation of these elements within the graphical user interface.

By dividing the project into these sections, I've aimed to create a modular and organized structure that facilitates development and maintenance while also ensuring clarity and ease of use.

The initial step involved constructing the GUI system. I began by crafting the foundational codes for Button, InputBox, and Window, as these elements were crucial for interacting with the simulation. To enhance flexibility and enable diverse behaviors, I leveraged the C++ functional library, enabling the passing of functions into constructors as parameters. This approach facilitated efficient code reuse and accommodated multiple functionalities.

Implementing the InputBox posed a greater challenge due to the requirement of simultaneously running the program and accepting user input. I adopted a threaded approach, wherein a separate thread managed user input, updating the data once input was complete. To prevent concurrent execution conflicts, I employed locks to ensure that only one InputBox could initiate a thread at a time.

The Window implementation was relatively straightforward, employing a vector to store all Element objects. This design choice allowed for the simulation of various components such as windows, buttons, and input boxes within the system.

The next step involved simulating physics. I created another GUI class called "PhysicsSim" which is responsible for storing, checking collisions, and updating all physics objects while keeping track of memory. To simplify the code, I chose to use only circles as collisions boil down to a simple radius check, and collision resolution could be achieved by merely moving them apart an equal distance. This approach also offered a performance gain, as conducting 2-3 iterations was sufficient to ensure that collisions would be resolved. The biggest challenge by far was ensuring that collisions and momentum transfer worked perfectly and preserved energy, I addressed this issue by calculation the before and after energies and adjusting to make sure they match as closely as possible.

Now, it was time to integrate physics into the system. I opted for Euler’s integration method to update the position, velocity, and acceleration of physical objects. This method seemed the most straightforward and efficient to implement. It also allowed for the separation of the graphics and physics update loops, reducing CPU load by computing every 0.003 seconds rather than every frame. Coupled with double buffering, this approach promised faster and smoother results by halving the workload. The next step was to implement physics for the joints.

Hinge Joint:

The calculation for the force followed these steps:

Determine the normal vector n from B (the object) to A (the anchor).

Calculate the perpendicular vector p.

Map the acceleration to the perpendicular vector and use it to update the force on B.

Calculate the force mapped onto the normal and subtract it from B to balance the forces.

Ensure that the object maintains a distance L from the anchor A.

Rigid Joint:

Using the XPBD algorithm, you can apply a force to keep the points A and B separated by distance L. This <https://carmencincotti.com/2022-08-08/xpbd-extended-position-based-dynamics/> goes into greater detail.

Spring Joint:

This is an implementation of Hooke’s Law: F = -kx. First relative position P from B to A, B and A are physical objects. Get the ||P|| - resting position. And apply the formula and damping.

Now that everything is put together all there is to do is initialize all components. First the windows get initialized, then, then the simulation window, and then all the buttons are initialized. The buttons are initialized via a lambda function, which allows the passing of parameters by reference. This allows for very flexible design by allowing the programmer to define behavior and interactions between objects.