**SimplePhysics**

A “can I do it” physics simulation library

The following details my plan for creating a simple, straight-forward physics simulation and visualization library. The physics “library” with which I am most familiar is Roblox, so SimplePhysics will very likely end up looking like a simplified version of that.

**Project Goal:**

This project is only meant to serve as an educational tool for myself. In other words, I’m just trying to learn how physics libraries work. Therefore, I will only be stressing getting it to work, not necessarily efficiency or usability. The final goal is to get this library to the point that I will be able to use it to develop a small game, such as Tanks (or something similar).

**General organizational setup**

Any interaction with this library will be done through what is called the “Workspace.” The Workspace is the basic container object that holds all simulation-dependent objects (collide-able objects, events, etc.) To start a new simulation, the user will create a new instance of it, add any simulation objects that the scene should start with, and then begin the simulation.

When the Workspace begins simulating the world, it will split off a separate thread which will be responsible for running the simulation loop. This will allow the original “master” script to continue to interact with the Workspace without interrupting the simulation loop.

**Simulation Loop:**

When a Workspace separates itself as a new thread, it will begin running what is called the “simulation loop.” Each iteration of this loop is equivalent to running a single simulation frame. It will consist of the following steps:

1. Event Collection:

Check through the entire simulation for any events that need to be signaled. Do not actually run the response to these events, simply look for them and notify the dependent objects.

1. Step the simulation:

Allow every object to respond to all events and decide their state for this simulation frame.

1. Render:

Draw every object’s new state.

**Object Hierarchy: Parenting**

Every object will have the ability to be “parented” to one Workspace. Objects have a plain-text, non-unique name that identifies that object in its Workspace. Each Workspace has a hash table containing every “child” object that that Workspace contains. When an object is parented to a Workspace, it will be inserted into that Workspace’s hash table, where the key is the object’s name and the value is the object itself. Because an object’s name is not necessarily unique, name collisions will be handled by having each hash table entry be an array of objects with the same name.

**Thread Synchronization:**

Semaphores will be used to synchronize access to the workspace between the simulation and main script threads. The Workspace will have a semaphore which will ensure that the next frame is not started if the main script is interacting with the Workspace. Each object will also have a semaphore that will control any direct-reference accesses.

**Rendering:**

Any object that can be rendered will be a descendant of a “Drawable” class. This class will provide a method that allows the given object to draw itself to the given Workspace’s window. When the Workspace hits the render step of its simulation loop, it will call this draw function.

**Event Collection:**

The event collection step of the simulation loop will consist of two parts. First, it will look for any user input (e.g, key presses and requests to exit the program). Then, it will allow every object in the workspace’s hash table to look for the events that it is dependent upon.

If an event is found in either stage, the callback for that event will be added to an “events” table for that object. Then, when the Workspace begins to calculate the next frame, each object will loop through its triggered events and run the corresponding callback.

Every object in the simulation (including the Workspace) will have an “events” hash table where the key is the event type and the value is an array of callbacks that should be run. It was decided to store the events in a hash table of arrays (rather than a single list of callbacks) so that certain events can be ignored based on the circumstances of the object. For example, if a collideable object is set so that it can’t actually participate in collisions, it will simply flush out all Collision events every frame, rather than using the actual callbacks.

**Event Collection: Registering a Callback**

Each simulation object (again including the Workspace) will have a method that will add a given function to the table of allowable callbacks (called the “callback table”). The callback table is a hash table indexed by the type of event with values being reference to the callback function. So, when an object fires an event, it will pull all applicable function calls from the callback table and add them to the object’s event table.