The Tethered Particle System and Associated Tools

Advanced Real-Time Simulation Laboratory - Carleton University
Thomas Roller (thomasroller@cmail.carleton.ca)
Winter 2021 / Summer 2021

Table of Contents

Tethered Particle System Preface

The Tethered Particle System (abbreviated TPS) is a Discrete Event (DEVS) based simulation of how particles interact. The authoritative document on the system's specifications, the thesis by Rhys Goldstein [1], details many aspects of the simulation. Although not every element of the original version of TPS has been included in the newer version of TPS, many of the fundamentals have been implemented or have some existing framework within the code.

This document will detail some of the basic theory behind TPS, how to compile the code, how to use the helper tools, and explain some features which were omitted from the current version. For a full description of the system's theory and feature set, refer to the original thesis.

The continued development of TPS requires that the user is comfortable with the DEVS formalism. An understanding of the DEVS simulator, Cadmium, is similarly necessary. While the information in this document will allow the user to obtain an understanding of TPS and how it functions, practical development will benefit from reading the available literature along with the in-code documentation.

It is important to know that the Cadmium implementation of TPS differs in some ways from the original specification. While the thesis is generally adhered to, certain elements of the model have been adapted to match the needs of both Cadmium and C++.

Basic Theory and Features

Collision Types

Fundamentally, TPS simulates particles moving through space in either one, two, or three dimensions. Particles move throughout the simulated space (which, in the system's current state, is infinite) and interact in with each other. The two primary types of interaction are blocking and tethering collisions. For the purposes of this implementation, only blocking collisions were considered as tethering introduced unneeded complexity.

Blocking collisions are what most people will think about when talking about particle collisions. This is when two particles come within a certain range of one another and ricochet (or bounce back). In the figure to the right, the distance between the centre of each particle in step 2 is referred to as the blocking distance. The blocking distance is typically calculated as being the sum of the radii of the involved particles.

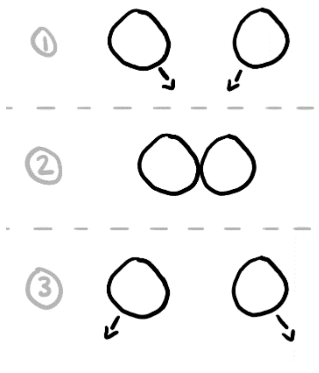


Figure 1: Blocking collision

Tethering collisions, while less intuitive, can be shown in an easy-to-understand manner. In the case of tethering, when the centres of the particles reach a certain distance, the tethering distance, they bounce back toward each other. This system can be conceptualized by attaching the tethered particles with a rope. In the first step of figure 2, the two particles are closer to each other than the tethering distance, resulting in the rope being slack. As the two particles move apart, the rope becomes taut. The particles will bounce back.

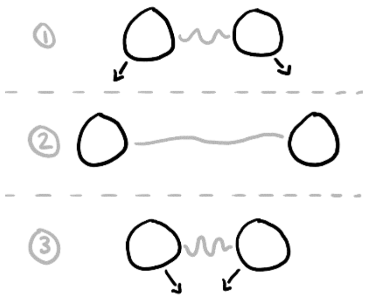


Figure 2: Tethering collision

It is important to note than figure 2 depicts a simple example of tethering. It is possible that two particles could begin rotating around each other given the proper initial velocities and positions. This situation is considered a special case and is handled appropriately. Details regarding tethering in this report will not extend past what is above as these collisions are not currently included as of the time of writing.

Modules and Messages

TPS consists of several distinct modules, each of which manage one or more aspects of the simulation. In order to communicate with one another, the modules send messages to each other. The content of each message depends on both who is sending and who is receiving the message. As a result, there are several different message types.

Random Impulse Module

The random impulse module (abbreviated "RI" in figure 3) is the simplest and most isolated module of TPS. Its job is to generate impulses for every particle in the simulation. These impulses are of random magnitude and direction. Given that this process requires no incoming information, the module only sends outbound messages. In figure 3, these messages are passed through the *impulse* port.

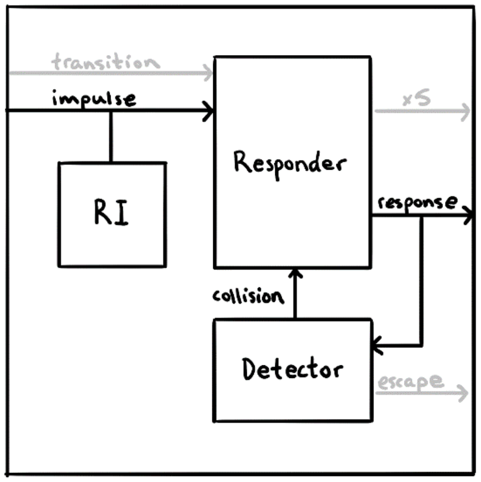


Figure 3: Modules and messages of TPS (grey indicates non-essential elements)

The module exists in TPS to nudge particles. There are several reasons for why this behaviour is beneficial to TPS simulations. First, if the user wishes to simulate particle's interacting with other particles (for example, air) that are not explicitly defined, the random impulse module can jostle particles. These small interactions can add realism to simulations.

Another, arguably more important, use for random impulses in TPS is to keep the overall momentum from trending toward zero. It is possible (and more realistic) for collisions to be at least somewhat inelastic. As a result, momentum can be lost during these interactions. Without intervention, particles in an enclosed space would become less energetic. By randomly nudging particles, we can effectively inject the simulation with momentum.

Given the nature of the module, it is possible to turn it off before the simulation begins. It is feasible that one would wish to avoid the unpredictability of the module in favour of a cleaner set of results. As a result, a setting exists within the scenario JSON to disable these impulses within the `config` section.

Responder Module

The responder module is responsible for managing all velocity-based tasks. These tasks include the following:

- Calculation and application of impulses from the random impulse module to the velocities of the affected particles
- Calculation and application of new velocities of particles after a collision has occurred
- Management of loading and restitution

The responder is critical to the function of TPS as it determines *how* particles collide. The way in which particles bounce off of each other is the responsibility of the responder. In order to do this effectively, the responder is in constant communication with the detector module (which handles *when* particles collide).

Since the module is so central to allowing TPS to function, it has a variety of messages being sent and received. The *response* port transports messages from the responder to the detector. These messages carry new velocities that the responder has calculated to the detector. They contain the velocity being sent and a list of particles which the velocity applies to. These messages are referred to as "velocity messages". The list of particles can contain a single value. In fact, without loading, the particle list will always have a size of one. These messages keep the detector up to date on the state of the model and allow it to calculate when and where the next collision will happen.

The two input ports, *impulse* and *collision*, relate to the random impulse module and the detector respectively. The impulse port is how the responder receives impulses to particles (many of which may come from the random impulse module). The *collision* port is how the responder is notified of collisions that require new velocities to be calculated.

Other ports on the responder include *transition*, *attachment*, *detachment*, *impulse*, *loading*, and *restitution*. Table 1 briefly outlines the purpose of each port.

Table 1: Messages associated with the responder module

Name	Type	Purpose
transition	IN	Affects how particles become tethered (unused)
impulse	IN	Impulses that modify the velocities of particles
collision	IN	Informs the responder about collision details
attachment	OUT	Used to log when particles are tethered (unused)
detachment	OUT	Used to log when particles are released from tethering (unused)
impulse	OUT	Used to log when a particle receives an impulse (unused)
loading	OUT	Used to log when a particle or loaded group loads (unused)
restitution	OUT	Used to log when a loaded group breaks apart (unused)
response	OUT	Notifies the detector of new velocities and is used to log changes in velocity

Detector Module

Like the responder module, the detector plays a critical role in TPS. Its responsibilities include:

- Calculation and storage of position information
- Calculation of the time of the next collision (and which particles are involved)
- Management of sub-volumes (current non-applicable)

The detector's primary job is to determine when the next collision will occur and between which particles by using every particle's position and velocity. In the case where there are limited optimizations (no caching and no divisions of the simulated space), every pair of particles in the model must have their collision time calculated. The detector then reports the next collision to the responder. These operations can take a significant amount of time. Through several optimizations, the detector can reduce the number of computationally expensive operations that must be done.

Three types of messages are used by the detector. When the detector determines when the next collision will happen, it sends a message to the responder so that it can calculate the resulting velocities. This type of message, sent along the *collision* port, contains the positions of the offending particles along with their IDs. These messages are referred to as "collision messages". The responder module uses this information to calculate velocities. These velocities are received through the *response* port using velocity messages. The final port, *escape*, is used when a particle leaves the simulated area.

Unlike the other modules, the detector is a coupled model. It consists of two models, one of which is also coupled. The coupled model, the lattice, is used when dividing the simulated space into several distinct regions called sub-volumes ("subV" in figure 4). The motivation behind these divisions is detailed in its own section. Although only one sub-volume is depicted in the figure to the right, when properly using this technique, there will be several. The dotted line indicates that there are messages being passed between different sub-volume modules. This port handles adjacency messages.

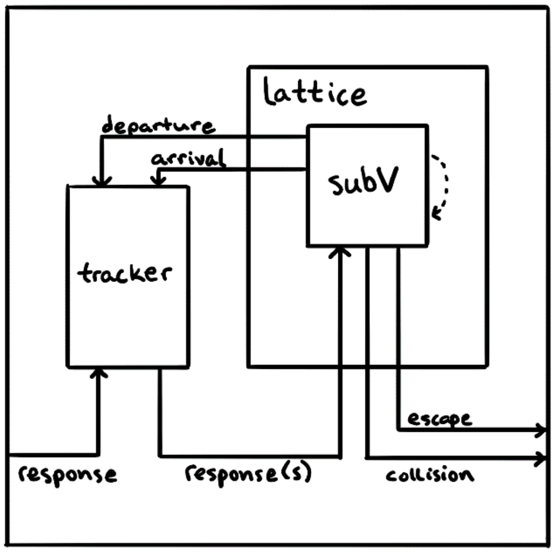


Figure 4: Structure of the detector module

The tracker atomic model has the responsibility of ensuring that velocity messages sent through the *response* port are routed to the correct sub-volume(s). If particle A is in sub-volumes 3 and 5, the tracker will label the message appropriately (i.e., adding the IDs of the sub-volumes to a vector) and send the data to all sub-volumes through the *response(s)* port. These modified velocity messages are called "tracker messages". Each sub-volume will then read the tracker's label to determine if the tracker message is relevant to itself. If the message is not meant for a sub-volume module, it will be discarded.

Table 2: Messages associated with the internal models of the detector

Name	Path	Purpose
response	D → T	Velocity messages arriving from the responder
responses(s)	T → L → S	Tracker messages destined for sub-volume(s) after being labelled
escape	S → L → D	Used to log when a particle leaves the simulated space
collision	S → L → D	Collision messages heading to the responder with collision data
departure	S → L → T	Messages to inform the tracker that a particle has left a sub-volume
arrival	S → L → T	Messages to inform the tracker that a particle has entered a sub-volume
adjacency	$S_A \rightarrow S_B$	Messages used by sub-volumes to keep track of neighbours

It should be noted that, in its current form, TPS contains one sub-volume with no boarders. The aforementioned models that make up the detector are all present. The lattice simply consists of one sub-volume. The *departure*, *arrival*, *adjacency*, and *escape* ports are yet to be included as they are not relevant when considering a single, unbounded sub-volume module. Despite this, certain elements of the system were designed in such a way as to make the addition of these features possible without significant adjustments being required. Some of the steps required to implementing further sub-volumes are detailed in another section.

Loading Optimization

Loading is an optimization to TPS which exists to reduce the number of collisions. Specifically, the goal is to limit the number of collisions happening in quick succession. A scenario in which this could happen is where there is a low-mass particle between two large-mass particles. The small particle could be in a situation where it is bouncing between the larger particles for a long time. Given that DEVS is event-driven, each collision requires new calculations and messages. If it is possible to avoid extraneous collision, TPS would receive a performance boost.

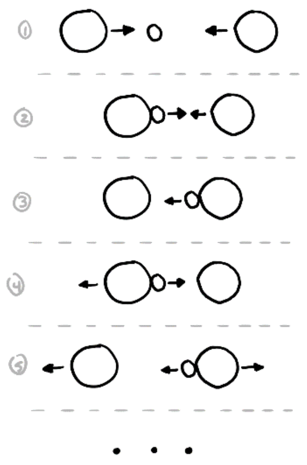


Figure 5: Loading motivation

The mechanism of loading works by effectively sticking particles together for a short period of time. When particles collide, they stay next to one another and move with the same velocity. After some period, called the restitution time, the particles break away from one other (restitution). In order to maintain conservation of momentum, the particles receive the other portion of the collision. This impulse (the restitution impulse) will send them moving away from other another.

The restitution impulse is the difference between the full impulse (what would have been applied if loading was not being used) and the impulse that would be applied to get the particles to move at the same velocity (the loading impulse). Although the velocity that the particles have while loading is calculated directly, there is a theoretical impulse that would cause that velocity. Once the restitution time elapses, the restitution impulse is applied to the particles/loaded groups involved.

Loading suffers from being an inexact way of handling many collisions. The larger a loaded group becomes, the bigger the hit to accuracy becomes. To counter this issue, the restitution time is generally very small which keeps loaded groups small. Despite loading not being perfect, the benefits outweigh the drawbacks. The high number of collisions being avoided grants TPS a significant edge over its loading-free counterpart. Even with its inaccuracies, the deviations that loading creates are not noticeable on most scales.

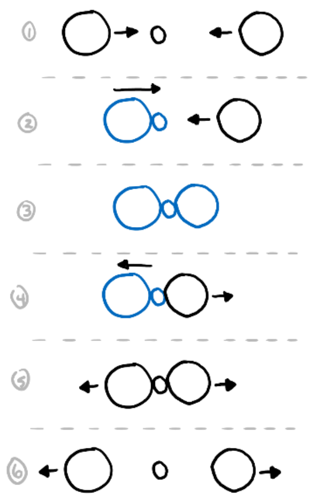


Figure 6: Loading example (blue particles are loaded)

Many of the equations used to facilitate loading are derived from the law of conservation of momentum. In fact, TPS's primary concern is maintaining this rule and loading is no exception. In order to conserve momentum, particles must be restituted in reverse order. Looking at figure 6, we see that the left two particles collide and load first. A restitution impulse is calculated and stored for these two particles. This group then collides with the rightmost particle, forming a loaded group of three particles. Another restitution impulse is calculated for this new loaded group. Once the restitution time elapses, the loaded group containing the two left particles and the rightmost particle (which can be considered a loaded group consisting of one particle) are restituted. The restitution impulse is applied to these two groups. Once the impulse is applied, we immediately restitute the remaining loaded group. The restitution impulse that was calculated for those two particles is applied and the particles move away from one another.

Usage Instructions

Required Software and Preparation

In order to compile and run TPS, there are several requirements. Much of the compiling and testing of TPS was conducted in Windows Subsystem for Linux 2 (WSL2) using Ubuntu 20.04 LTS. Although software such as Cygwin should function, that is not the environment in which the model was tested. Within WSL2, g++ (2017) was used to compile the program.

The C++ library Boost is used to provide hashing for pairs. It may be necessary to include Boost in the Makefile as its location may depend on each user. Another library for parsing JSON files is used by TPS extensively. It is necessary to include this library in the Makefile as well.

TPS is built on the Cadmium simulator and requires it to compile and run. TPS was built while it was located in the directory `Cadmium-Simulation-Environment/DEVS-Models`. It may be necessary to modify the Makefile's include statements to account for where TPS is located.

Detailed below are the steps to preparing the required environment and software for compiling and running TPS.

Set up Cadmium. Details on how to accomplish this task can be found in Cadmium's installation manual, here [2]: http://www.sce.carleton.ca/courses/sysc-5104/lib/exe/fetch.php?media=cadmium.pdf. Note that these instructions detail how to prepare Cygwin, not WSL2. Should you opt to use WSL2, you will need to obtain Boost manually and include it in TPS's Makefile.

To use Boost, download the library from its website: https://www.boost.org/users/history/version_1_77_0.html. Both Cygwin and WSL2 should use the Unix download. Extract the files using the command:

```
tar -xf boost_1_77_0.tar.gz.
```

In TPS's Makefile, include the directory: `path_to_boost_extraction/boost_1_77_0/boost`. The library is used in the `subV.hpp` file.

Compilation and Execution

Run the following command to compile TPS using the provided Makefile:

```
make iter_1
```

The executable that is created (located in the `bin` directory) takes one argument which specifies where the scenario JSON is located. This file describes the initial conditions of the simulation such as particle locations and velocities.

Before running the program, make sure that there is a folder called `simulation_results` on the top level of the TPS directory structure (alongside `atomics`, `top_model`, and others). This folder is where simulation log files will be placed.

To run TPS, navigate to the `bin` directory and enter the following command:

```
./ITER_1_TEST [scenario JSON]
```

Using Cygwin, it may be necessary to append the executable's name with ".exe".

Implementation Details

TPS is a modular system that divides responsibilities among several different components. As a result, each module can be described in terms of its duties and communications with other elements. This section explains the code of each section. For the responder and subV modules, this follows the code closely.

Random Impulse

Primary file: `atomics/random_impulse.hpp`

Currently, the module makes use of a priority queue to store and manage when the next impulse should be delivered. Once an impulse is sent to the responder module, the item is popped from the queue and the next particle will be set to receive an impulse.

Responder

Primary file: `atomics/responder.hpp`

External Transition

The responder module operates primarily based on inputs. When a collision is received from the subV module, the external transition function first gathers and organizes information regarding the involved particles. Although a collision message will always contain exactly two particles, the particles considered by the responder will include entire loaded groups.

Once the necessary information is obtained, impulses and velocities will be calculated. The values that are created here include: the full impulse, the loading impulse, the loading velocity, and the restitution impulse. Before any of these values can be used, we need to update the list of currently loaded particles (which is stored in a state variable called `loading_trees`). We first determine which nodes are relevant for the new collision (if there are any at all). A new node is created to represent the current collision. With this, these nodes are added to the newly created node as children and removed from `loading_trees`. Every descendant particle's restitution time is updated to match that of the parent.

The responder then updates its stored velocities to the loading velocity that was calculated earlier. Before the external transition function is complete, a message containing the new velocity is sent to the detector in order to keep the other modules updated. Finally, the time until the next internal transition is set to zero so that the messages can be sent immediately.

Internal Transition

Once the waiting period has elapsed, the internal transition function is called. In contrast to the external transition function, this function sets the time until the next internal event almost immediately after being called. This change is necessary since, if the last external event was due to a random impulse, we return before much of the function has a chance to execute. The next internal event will happen when the next node is ready to restitute.

A state variable, `buffer`, is checked to see if it is not null. This variable stores a pointer to the node which is to be restituted next and is set later in this function. By default, it is null. If the value is not null (i.e., this function has already been run and there are nodes which can be restituted), we enter into a conditional block of code where the responder performs the restitution of that node. The child nodes of the `buffer` node are added to the `loading_trees` set and the `buffer` node is then removed from the set. The new velocities for the particles are applied to the responder's state from previously set messages. The now restituted node is deleted from the heap and the pointer is reset to null. This marks the end of the conditional block of code.

Since the set of particles that need to be restituted has been updated, it is necessary to recalculate the time until the next internal event. The aforementioned messages are cleared to prepare for the next set of messages (which were just used earlier in the conditional block from the previous run of this function).

If there is at least one node in `loading_trees`, we calculate the velocities for the next restitution. First, we obtain the node that will restitute next and keep a copy of the pointer in the `buffer` variable. In a similar way to how we gathered information in the external transition function, we gather information on the involved particles and their loading groups. Using the restitution impulse that was calculated in the external transition function (and stored in the node), the velocities of the involved particles are calculated. Messages containing the relevant data are then created. These messages are used to both send messages to the detector and update the responder's internal model when the next internal event occurs.

Tracker

Primary file: `atomics/tracker.hpp`

The tracker's job is to receive messages from the responder and direct them to the appropriate sub-volume(s). Although there is currently only one sub-volume, the tracker will send any messages it receives to every sub-volume. With each message that it receives, the tracker attaches a list of sub-volumes that the message is destined for. Every sub-volume will get every message and it is each sub-volume's responsibility to read the tracker's addendum and determine if a message is relevant to it.

The tracker is set to always send messages to the sub-volume with ID #1 since there is only one sub-volume. The tracker is capable of sending messages where they are needed (although the line that does this is commented out). However, there is currently no logic to initiate/update the tracker's knowledge of which sub-volume each particle is located in.

Sub-Volume

Primary file: `atomics/subV.hpp`

Internal Transition

The sub-volume's internal transition function begins by setting a flag which notes that is it prepared to send a collision message to the responder. It also clears any previously created messages. If the responder has just sent out a collision message and is waiting to hear from the responder with new velocities, it will passivate itself and wait for a message from the responder. Assuming the sub-volume has already received the necessary messages, it will then update its internal cache of collisions to incorporate the previous collision. After this check, it will calculate when the next collision will happen and with which particles. It is worth noting that this process does not pay any attention to which particles are loaded and only concerns itself with individual particles.

The sub-volume then calculates the positions that the colliding particles will have but does not immediately incorporate them. Instead, the data is stored in a state variable called `next_collision`. This is the message that is prepared for the responder. The information cannot be applied right away in case a random impulse is received before the collision messages are sent to the responder. If a random impulse is received, the collision data is thrown away since the new impulse may alter when the next collision will happen and with which particles. If no unexpected impulses are received, the next internal event is scheduled and a flag is set to note that the sub-volume requires velocities from the responder.

External Transition

When velocities are received (whether they be from the responder or random impulse module), several flags are set to ensure that proper timing is maintained. The sub-volume then checks each message that is received to see if the tracker included its ID in the message. If the message is for the sub-volume, it notes that it has received an applicable message. For every particle that is included in the message, the position of the particle is updated. A note is made for each particle, denoting the time when the particle received its new position. These notes are extremely important when calculating new positions for particles.

The velocities for the particles are updated to reflect the information in the received messages. Messages are then prepared for logging purposes. These messages are not sent to any module in particular. Finally, the next internal event set to happen immediately so that the next collision can be calculated.

Visualizer

Primary file	`visualizer/main.py`
Default configuration file	`visualizer/config.json`

The TPS visualizer is a developer tool that can be used to display the state of TPS at each event in the simulation. The user can step through simulation collisions, particle restitutions, and random impulse applications using the provided user interface. The program does not use any third-party modules and relies entirely upon features that come installed with Python 3. However, the program does require a way to display graphics. WSL2 (by default) does not provide this ability. The visualizer was developed using Windows PowerShell.

Since the visualizer steps through events in the simulation, this means that only the particles involved with a particular event will be updated. TPS only updates a particle's position when its velocity is changed. The new position is calculated based on the previous velocity and how long that velocity was held. As a result, TPS does not provide any new information for the visualizer to display outside of events involving those particular particles. While it is possible to interpolate positions based on the data given by TPS, that is outside of the scope of this visualizer.

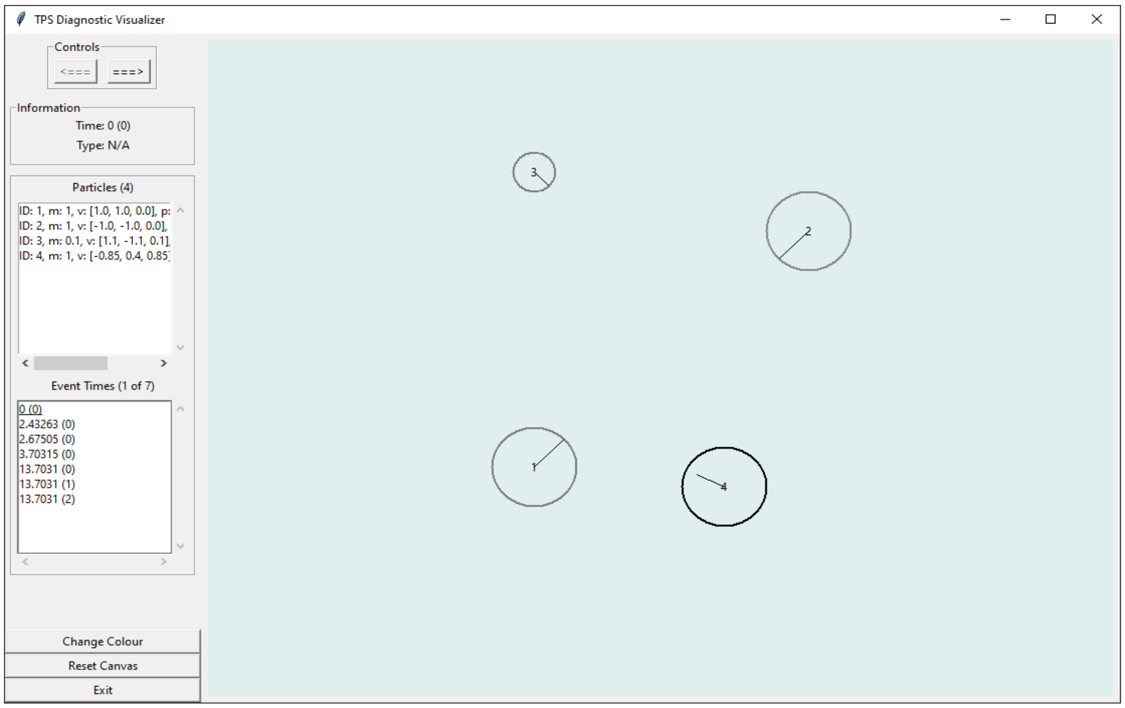
Configuration

The visualizer pulls data from several different places. These files are specified in the visualizer's configuration file. Below is a table outlining each parameter in the configuration file.

Table 3: Configuration details for the visualizer

Setting	Purpose
`messages`	Messages log generated by Cadmium
`config`	Scenario JSON used to run TPS
`simulation`	Rough dimensions of simulated space (expecting a list of two values, [x, y])
`visualizer`	Dimensions of visualizer canvas (in pixels)

Usage



```
Time: 0 (0)
Type: N/A
```

```
Particles (4)
ID: 1, m: 1, v: [1.0, 1.0, 0.0], p:
ID: 2, m: 1, v: [-1.0, -1.0, 0.0],
ID: 3, m: 0.1, v: [1.1, -1.1, 0.1],
ID: 4, m: 1, v: [-0.85, 0.4, 0.85]
```

```
Event Times (1 of 7)
0 (0)
2.43263 (0)
2.67505 (0)
3.70315 (0)
13.7031 (0)
13.7031 (1)
13.7031 (2)
```

Figure 7: Screenshot of the visualizer program displaying simulation results

In the `visualizer` directory, run the command:

```
python3 main.py [config]
```

The program's only argument is the configuration file. If the argument is left blank, the default will be used.

In the visualizer, the main control mechanisms are the two arrows at the top left of the window. These arrows allow the user to navigate forward and backward through the simulation. Immediately below the arrow controls is information about the current event such as its time, type, and involved particles. Below the information box are two tables. The table labelled "Particles" provides details about every particle at the current time. The particle's ID, mass, velocity, and position are listed here. The table labelled "Event Times" shows all of the events that TPS generated. The number in brackets helps differentiate between events that happened simultaneously (which is especially relevant with the loading optimization). The user can click on an event in this list to skip to it. The current event is underlined.

There are three buttons on the bottom left. The "Change Colour" button cycles the colour of the canvas. If some particles are hard to see, changing this setting may help. The "Reset Canvas" button attempts to recenter/rescale the canvas. The behaviour of this button can be unreliable at times. Finally, the "Exit" button closes the application.

The majority of the program's window is occupied by the canvas. The user can click/drag and scroll to navigate this area. Should the user be viewing a 3-dimensional simulation, the position of each particle in the Z-coordinate is represented through shading. The more positive the coordinate (closer to the viewer), the darker the particle's outline becomes. Conversely, the more negative the value, the lighter the outline becomes.

Scripts and Helper Programs

During the development of TPS, it became necessary to create scripts that could assist in data parsing and visualization.

Generate Random Scenarios

Primary file	`input/generateRandom.py`
Default configuration file	`input/config_template.json`

While it makes sense to manually input particle information for small scenarios with only several particles, the process of entering data manually can become tedious for creating a large scenario for testing purposes. The simple program, generateRandom, can randomly place non-overlapping particles. It is able to accomplish what would take a user a significant amount of time.

The script takes one argument on the command line. The file that it reads is a JSON that represents a scenario but with no particles. The primary purpose for this file is to provide the script with the possible species that it can use to randomize particles. If this argument is not provided, the program uses its default configuration file. Run the following command to use the script:

```
python3 generateRandom.py [config]
```

The program works by placing a particle in a random place and then checking that it does not collide with other particles. If there is overlap, the new particle is discarded and the program tries again. By default, it will try 10 times before ending the program. The number of attempts is a keyword argument of the function `genParticles`. When the maximum number of attempts is reached, the script will export what it was able to generate before the failure. This problem can be mitigated by providing the program sufficient space to place particles. Near the bottom of the program, it is possible to adjust how much space is allotted by modifying the variable `posRange`.

Despite the fact that the user is not required to provide any arguments, the behaviour of the script is very customizable. In order to modify settings, the user must change the values given to certain variables within the program. Namely, the variables under the comment “Configure and execute the script” near the end of the code.

Generate Graph of Cache Size

Primary file	`extras/cache_size/generateGraph.py`
Default data file	`extras/cache_size/cache_size.txt`

In order for TPS to know when the next collision will occur, it needs to perform many calculations. Each pair of particles in the simulation needs to be checked. The naïve approach to this problem is to calculate the time until every pair of particles will collide. With this method, particles that are unaltered will be recalculated after each collision. In order to avoid an excessive number of unnecessary calculations, a cache is used to store unaffected particles. The size of this cache changes over time since particle that are determined to never collide are not stored. As it is implemented, TPS benefits from a smaller cache.

The script, generateGraph, exists to graph the size of the collision cache over time. This program served primarily as a development tool to help visualize how the size of the cache changes. Generally, the number of collisions in a simulation with one, unbounded sub-volume will decrease. With bounded sub-volumes, this may no longer be the case as we would have to account for particles colliding with boundaries (although this may be done using a separate cache).

Running the program requires that the `plotly.express` module be installed. To use pip to install the module, enter one of the following commands:

```
pip3 install plotly.express
python3 -m pip install express
```

The above commands should also install another required module: `pandas`.

The script takes a file containing the cache size as its only argument. The file is generated through a series of messages to the terminal. To generate the required file, it is necessary to change several flags in the `tags.hpp` file. Namely, every flag should be false except for `CACHE_LOGGING` which must be true. Once these flags are set, TPS must be recompiled.

Generating the data file is simply a matter of piping the program's terminal output to a file:

```
./ITER_1_TEST > cache_size.txt
```

Either move the created file to the same directory as the generateGraph script or specify the new file in the argument. To run the script, navigate to the script's directory. Run the command:

```
python3 generateGraph.py [cache size data]
```

The graph should appear in a browser window.

Future Additions

Below are some features that have not yet been added to TPS. Both "Walls and Immovable Objects" and "Gravity and Airflow" are generally outside of the original specification of TPS and have little to no backing within the guiding thesis.

Multiple Sub-Volumes

Refer to p43 and onward for more information regarding sub-volumes and their underlying theory.

A feature that is detailed in Goldstein's thesis that is not implemented in the current iteration of TPS is sub-volumes. Although some of the framework exists to facilitate the inclusion of multiple sub-volumes, TPS currently uses a single infinite region. The benefit to having several different regions is that the number of particles that must be checked is reduced.

If the simulation has n particles, it is necessary to preform $\mathbb{C}_2^n$ calculations. This number can be reduced by using caching (which is the case with TPS). However, it is necessary to perform this number of calculations at least once. Even so, at least $2n$ calculations must be done every time a collision occurs (when using caching). These numbers suppose that there is only one sub-volume. By splitting the simulated region into multiple sub-volumes, n refers to the number of particles in each sub-volume. Overall, the number of required calculations is reduced.

For example, if we have two sub-volumes in which the particles are evenly distributed, the number of initial calculations required is no longer $\mathbb{C}_2^n$ but instead $\mathbb{C}_2^{n/2} + \mathbb{C}_2^{n/2}$. Each additional sub-volume further reduces the number of collisions that must be calculated (again, assuming an equal distribution of particles among the divisions).

In order to manage these additional regions, several changes to TPS would be necessary. Some notable changes have to be made to the tracker. The code which labels each message with the relevant subV module(s) already exists within the programming. However, the line that actually labels each message is commented out in favour of a line that labels each message with sub-volume ID #1. Even with this change reversed, it would be necessary to properly initialize the tracker's internal knowledge about where each particle is located. Ensuring the tracker stays updated as particles move between different regions would also require the implementation of new ports and messages. Namely, the *departure* and *arrival* ports would be responsible for informing the tracker about particles moving from one sub-volume to another.

While the tracker would require modification, the subV module itself would need to be updated. According to the TPS's guiding thesis, adjacent sub-volumes are capable of passing *adj* messages between one another. This mechanism exists to ensure sub-volumes know when a particle is moving between regions.

Walls and Immovable Objects

In its current form, TPS permits arbitrarily massive particles. While there is no way to define a particle with infinite mass, a sufficiently massive particle would serve a similar purpose over a limited time frame. Unfortunately, these types of particles would not serve as a good replacement for something such as a wall for several reasons. Firstly, TPS assumes that every particle being simulated is a point with a constant radius (i.e., a circle or sphere). In theory, many of these massive particles could be given small radii and placed next to one another.

This workaround for walls leads us into the next problem which is that these blockades would still be particles. TPS would still make an effort to check every particle in the de facto wall against every particle in each sub-volume. Given a sufficient number of particles in the wall, the number of calculations required would be highly excessive for the desired goal. One possible way to prevent these particles from causing such problems would be to have a set containing the IDs of these particles and have TPS ignore them when checking for collisions. However, this is a hack of sorts and it not ideal.

In order to efficiently and elegantly include immovable objects, a new type of item would need to be added to TPS.

Gravity and Airflow

Including gravity in TPS is possible through a variety of ways. A rudimentary way of simulating gravity would be to apply frequent impulses in the downward direction. Although this would cause the particles to move downward, this would not be an ideal system as it would require a significant number of impulses for particles to follow a trajectory that appears smooth. It would be feasible to modify the way in which positions and collisions are calculated in order to simulate gravity. This approach would be much cleaner than using impulses but would also require a reworking of existing systems.

Airflow faces similar issues to gravity. As such, the solution to airflow in TPS may resemble that of the solution to gravity in TPS. If a global airflow is desired (i.e., every particle is affected), modifying the mathematics behind position and collision calculations would likely be sufficient. However, having a regional airflow (such as near an open window) would be much more complex since not every particle would be subject to the moving air. Despite these complications, it is within the realm of possibility to have different position functions (or perhaps a parameterized position function) which behaves differently in certain areas.

These adjustments to TPS are likely both possible, albeit with some major changes made to some core components of the system.

Adjustments

Many of the values that influence the way in which a simulation will run are specified in the scenario JSON file. Despite this, there are some values that are hardcoded in TPS. Ideally, these values should be calculated using existing or new values in the JSON. There are two major values that this applies to within the code. Values that should be calculated are preceded with the comment:

```
// TODO: should be calculated
```

The values are `restitution_time` and `c_restitute`, both of which are found within the responder module. The former is responsible for how long it takes a loaded group to release from when the last particle loaded. The latter determines how elastic collisions are in TPS. A restitution constant of one means that the collision is fully elastic and particles will not lose any energy. The closer the restitution constant gets to zero, the more inelastic the collision becomes.

Known Issues

Loading in TPS makes use of dynamic memory allocation. In order to prevent associated issues from manifesting, pointers are handled with care in the code. Unfortunately, for more complex simulations (for example, 1000 particles), it is possible to get the following set of messages:

```
free(): invalid pointer
        Aborted
```

TPS only has one call to the `delete` keyword. In order to diagnose the issue, this line was commented out since the error appears to be caused by a failed attempt at freeing memory from the heap. Even with this line removed, the problem persisted. While it is possible that there are other hidden issues within TPS, it seems that these messages are caused by code outside of TPS.

DEVS Considerations

It is worth discussing some basic DEVS features. Specifically, DEVS is only concerned with discrete events. In other words, the time between said events is discarded. For TPS, the most notable events are particle collisions. Barring other events (such as random impulses), the simulator will effectively skip over the time between these collisions.

If we have one collision five seconds into the simulation and then another at ten seconds, the simulator will jump forward to the next collision. In order to make such leaps, the program needs to be constantly aware of when the next collision will occur. In order to accomplish this, it must check the time until every particle will collide with every other particle. The computational effort that is required can be reduced by using sub-volumes. Refer to the Future Additions section for more on sub-volumes.

References

[1] Goldstein, Rhys. *DEVS-Based Dynamic Simulation of Deformable Biological Structures*. Carleton University, Sept. 2009, http://cell-devs.sce.carleton.ca/publications/2009/Gol09/

[2] “Cadmium – A tool for DEVS Modeling and Simulation – User’s Guide.” Internet: http://www.sce.carleton.ca/courses/sysc-5104/lib/exe/fetch.php?media=cadmium.pdf