A report on CRAPPY (the Cool Realism-Adjacent Physics Package, Y'know?) and the game I produced with it.

Or 'Misadventures in Mismanaged Mathematics'

Abstract

In which I discuss 'CRAPPY' (the Cool Realism-Adjacent Physics Package, Y'know?), along with 'A Scientific Interpretation of Daily Life in the Space Towing Industry circa 3052 CE', a game which I produced using CRAPPY.

Rachel Lowe; student no. 2100816

Rl18730@essex.ac.uk

Contents

Game Description

A Scientific Interpretation of Daily Life in the Space Towing Industry Circa 3052 CE (or, for the sake of everyone's sanity, *SpaceTow*), is, in short, a somewhat simplified clone of *Thrust* [1], in which the player controls a spaceship which must fly into a cave, find a payload, and tow it out of the cave, whilst trying to not crash into the walls of the cave or allow the payload to hit the walls of the cave, whilst trying to not fall foul of the laws of physics. It probably counts as being of the 'arcade'/'action' genres.

Gameplay is relatively simple. The player can apply torque to rotate their ship clockwise by holding left or D, apply anticlockwise torque by holding right or A, and apply linear force in the direction their ship is currently pointing in by holding up or W. When the player's ship is close to the payload, they can attempt to start towing it by pressing space; if the ship is close enough, a mostly rigid connector will be created between the ship and the payload, and the payload will unfreeze, allowing it to be towed by (and potentially interfere with the physics of) the player's ship.

There are only three levels, but the sound effects, graphics, and (most of) the codebase (including the physics) were overengineered from scratch, which probably counts towards something. Additionally, the player can easily pause the game (and get asked if they wanted to quit the game) by pressing the escape key (or attempting to close the game window) whenever. Furthermore, to reassure a player that some effort was put into making the user experience for this game somewhat bearable, the game window has a nice custom icon instead of the default icon, and the game window can be freely resized, with the viewable game being rescaled appropriately, to ensure that the game is at least playable at any window size.

Comparison to *Thrust*

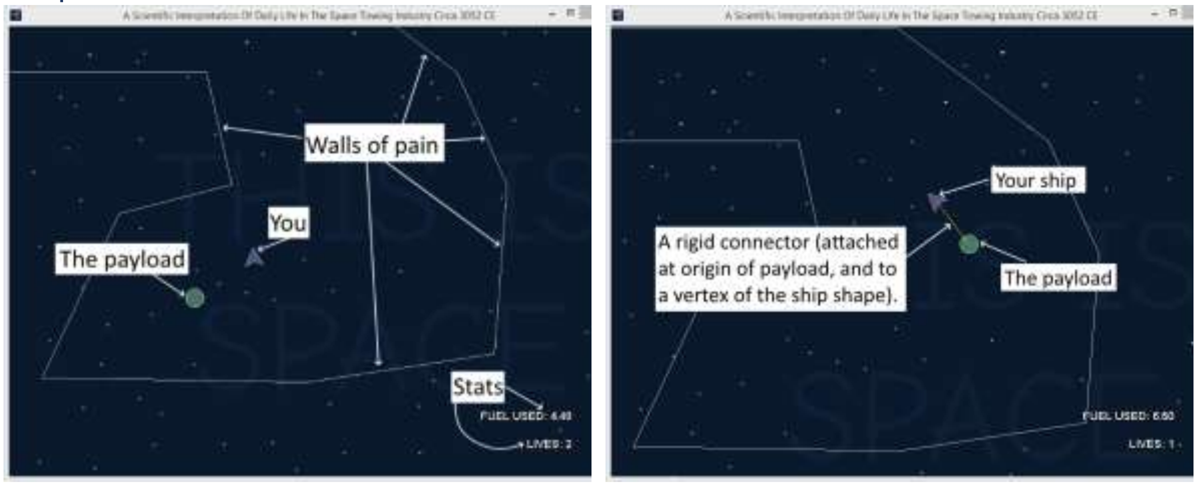

Figure 1: A couple of annotated screenshots of 'A Scientific Interpretation of Daily Life in the Space Towing Industry Circa 3052 CE'

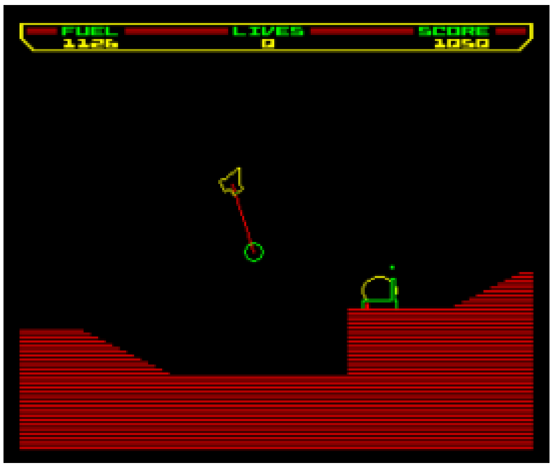

Figure 2: A screenshot of 'Thrust' , demonstrating the ship towing a power pod, starting to fly off away from the planet, with the HUD visible as well

Whilst *Thrust* has 6 levels (which looped with variations on them), *SpaceTow* currently only has 3 levels (ending after the third one), so there isn't quite as much replayability. However, there is a relatively simple process for creating new levels. One merely needs to define the static level geometry (taking advantage of a factory method to convert a flat collection of CrappyBody objects into a QuadTree data structure), a background image, and the start positions of the ship/payload, edit the 'levels' enumeration to add a new member to it capable of generating that level (following the same process as how the other members of that enum generate the other levels), and that's it. However, level 3 does differ a bit from the other two levels (and significantly from *Thrust*) by incorporating some unexpected ergodic literature. Other omissions from *Thrust* include the enemies, combat, barriers, and a hard limit on fuel (whilst *SpaceTow* does include a fuel counter, I wasn't sure how much fuel would be a reasonable amount to expect a player to need in a level, so I decided to leave it as a counter of 'total fuel used' so a player can try to minimize it if they want to). Additionally, whilst a level of *Thrust* is completed by flying up, away from the level, the levels in *SpaceTow* are fully enclosed, but are instead won by bringing the payload past an invisible 'finish line', near where the player started (where, instead of flying away, the player is told that they won, as they watch the connector between the ship and the payload break, and the laws of physics taking control of the movements of both).

One minor difference between the physics of *SpaceTow* and *Thrust* regards the pods/payloads that the player must retrieve. In both games, the cargo is connected to the ship via a rigid connector, and is connected to the centroid of the cargo. However, in *Thrust*, the connector connects to the centroid of the ship, whilst in *SpaceTow*, the connector is intentionally not connected to the centroid. This non-centred connection point on the ship in *SpaceTow* means that the payload's movements will result in linear force and torque to be applied to the ship via the connector, meaning that the player will need to exercise additional caution when towing the payload in comparison to *Thrust*, lest they accidentally jacknife their ship.

Technical Issues

The physics engine – *CRAPPY*

To be completely blunt, the main reason why I chose to implement my own physics engine from scratch instead of taking the sensible option of using JBox2D was simply because I wanted to use the name '*CRAPPY*' (Cool Realism-Adjacent Physics Package, Y'know?) because I thought that giving it that name would be funny.

That said, it does do a few things that JBox2D doesn't really do, such as supporting double-precision floating point numbers (instead of mere floats), offering immutable vectors (avoiding any unexpected side effects of some vector arithmetic) whist still providing mutable vectors (for people who may be concerned about garbage collection), and plenty of interfaces (for any programmers who may prefer using interfaces to using the classes themselves).

Despite this, it does still have several limitations, such as not supporting compound shapes, not having any joints besides an elastic connector, no raycasting, and, overall, a lot less polish.

Features of *CRAPPY* used by my game

Rigidbodies

This game takes advantage of *CRAPPY*'s rigidbody (lovingly referred to as 'CrappyBody') system. *CRAPPY* supports static, dynamic, and kinematic bodies (static bodies refuse to accept any forces, kinematic bodies refuse to accept any forces given to them internally via the engine but accept forces applied 'manually' by the programmer (via calling the appropriate methods), whilst dynamic bodies accept forces from the engine and the programmer), however, whilst I was able to incorporate static and dynamic bodies into *SpaceTow*, I wasn't able to think of a way of tastefully shoehorning in a kinematic body. Furthermore, there are four types of collision shapes present within *CRAPPY*: Circles, Edges (one-way walls, which can have a 'depth', and have a circle collider at the end of them to avoid clipping through corners), Lines (two-way walls, via having two zero-depth edges in different directions on top of each other), and Polygons (An arbitrary polygon, described by a list of vertex Vect2Ds, and concave shapes are not explicitly unsupported), all of which have automatically calculated moments of inertia based on rotating about (0,0) in local coordinates (taking advantage of the parallel axis theorem when the defined shape isn't centred around (0,0). Circles and polygons can safely collide against each other and any other shapes, but, whilst there is code for line/line, edge/edge, and edge/line collisions, colliding them is discouraged because those shapes are intended to be used as static geometry (maybe even kinematic geometry), not dynamic. The payload and the ship take advantage of how *CRAPPY* also allows rigidbodies to have their position/orientation temporarily frozen (the ship being frozen until the player starts pressing the direction buttons, and the payload being frozen until the ship crashes into it, or starts towing it). The invisible 'finish line' in the static geometry takes advantage of how *CRAPPY* allows rigidbodies to be marked 'intangible'; these intangible bodies can check for collisions like any other body, but, if at least one body involved in a collision is intangible, no forces will be applied for that collision, but any appropriate callbacks will still be sent.

CRAPPY also includes a 'layer'/'tag' system for these bodies, via bitmask abuse. Each body can have 'tags' (in the form of an int interpreted as a bitmask of 0s and 1s), and a bitmask indicating which tagged bodies it is allowed to collide with (only being allowed to collide with an object where at least one '1' is shared between their bitmasks, or being allowed to collide with anything if the 'can collide with' value is negative). Within *SpaceTow*, the ship may only collide with the walls of the world and the payload, the payload may only collide with the ship, the walls, and the finish line, the cosmetic debris may only collide with each other or the walls, the intangible invisible finish line may only collide with the payload, whilst the walls and may interact with anything besides themselves and the finish line. I will reiterate that the cosmetic debris (spawned in after the payload or ship are destroyed) is only allowed to collide with other debris and the walls, as, if one was not aware of that, one may think that the inability for the cosmetic debris to collide with the non-cosmetic ship and payload is a bug.

CRAPPY also has a callback system for collisions. There is a 'CrappyCallbackHandler' interface, which has three methods; one which passes a view-only interface for the other CrappyBody that the current body just collided with (called immediately after any necessary force applications on each body involved in a collision, if the bodies actually did collide), one which passes a bitmask which is the result of combining (via OR) the tag bits of every body that this body collided with in this collision detection iteration (called after each individual collision has been handled), along with a method which notifies the callback handler when *CRAPPY* removes the body from the world. This follows a similar principle to how Unity's MonoBehaviour class contains callbacks for quite a lot of engine events. The 'ship' and 'payload' objects implement this interface, and these notify the game level object if they have crashed into a wall/each other/if the payload has crossed the finish line, in turn notifying the game level about it, marking the ship/payload body for removal (if necessary), and telling the level to spawn in the debris (if necessary). This callback system could be expanded upon in the future to incorporate further callbacks (perhaps an 'on update' callback (maybe even a 'on sub-update' callback), but, for the time being, it works.

A basic rendering system, intended for ease of prototyping

CRAPPY offers an interface for rendering rudimentary graphics, via the 'I_CrappilyDrawStuff' interface, inspired in principle by Box2D's 'b2DebugDraw' class [2]. In short, the interface has some abstract methods for drawing simple shapes (given basic information such as positions, start points, end points, radii, etc), which the programmer needs to implement themselves (along with providing an implementation of an 'IGraphicsTransform' object, which it uses itself to scale/translate the coordinates from world scale to the intended screen locations, for ease of implementation. The programmer may also tweak whether or not they want to show the axis-aligned bounding boxes, velocity lines, orientation lines (which way is 'up' for a body), and polygon incircles (the largest possible circle contained entirely within the bounds of a polygon shape, centred around the centroid of it), by overriding a few more methods in this interface to return true/false (default implementations returning true), to allow these to be toggled in a way that's slightly more elegant than a couple of static boolean variables in the interface. 'I_CrappilyDrawStuff' is not intended to be used as the rendering method of choice for *CRAPPY*, instead being mainly intended for quick-and-dirty graphics rendering for debugging etc, however, I ultimately did rely on 'I_CrappilyDrawStuff' for rendering the physics world within *SpaceTow* (with the HUD text and background images being rendered via dedicated logic outside of *CRAPPY* itself). Finally, for full disclosure, whilst the abstract methods of and principle behind 'I_CrappilyDrawStuff' was heavily inspired by Box2D's 'b2DebugDraw' [3], the actual business logic within 'I_CrappilyDrawStuff' was made entirely from scratch, not referencing any external source code. A screenshot of *SpaceTow*, with the renderer settings adjusted to show this debug information, can be seen on the following page.

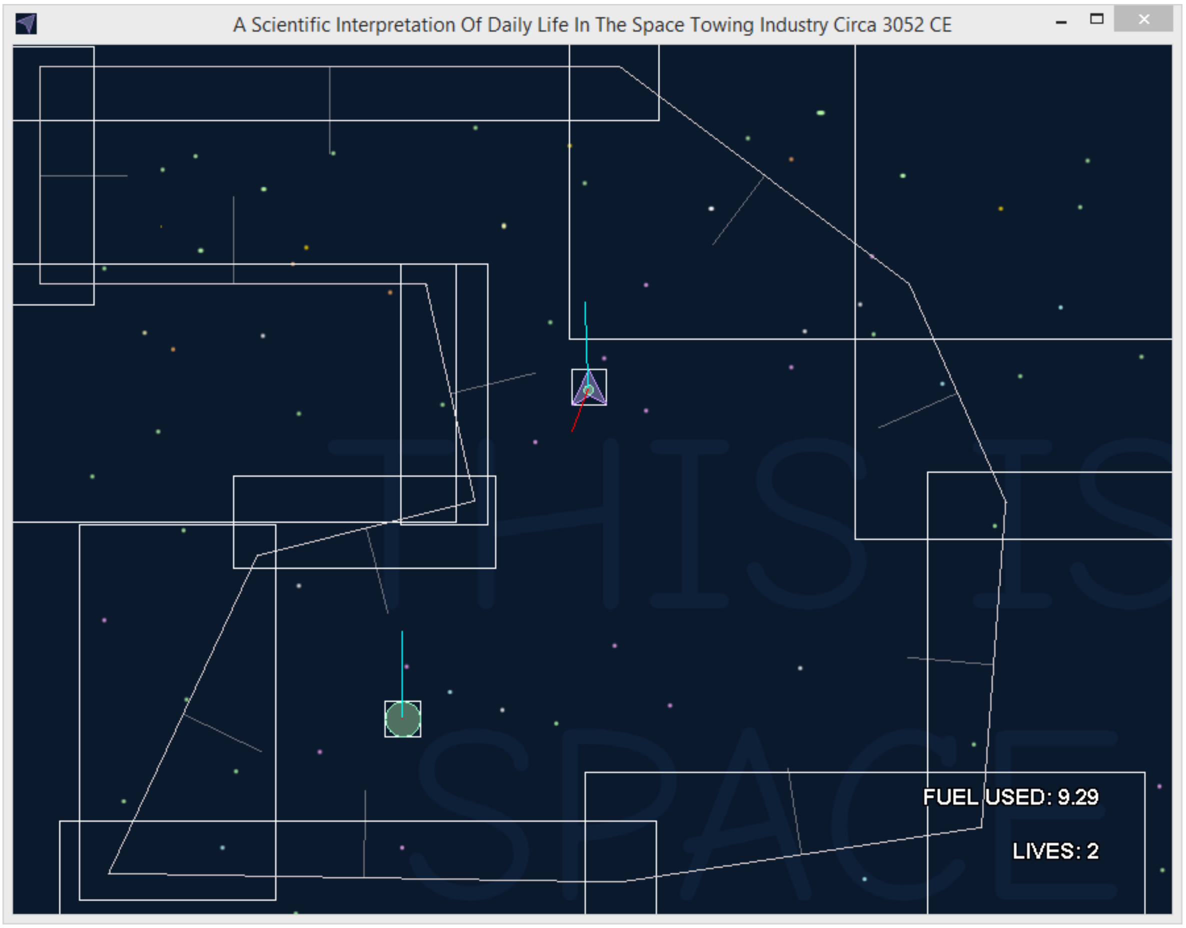
A Scientific Interpretation Of Daily Life In The Space Towing Industry Circa 3052 CE
THIS IS SPACE
FUEL USED: 9.29
LIVES: 2

Figure 3: A screenshot of level 1 of SpaceTow, but with the renderer settings edited to show all of the debug information (such as the bounding boxes (in white), edge normals (in grey), polygon incircles, velocity lines (in red), and orientation lines (in blue).

The CrappyWorld, and its customizable update loop

All of the CrappyBodies (and the CrappyConnectors potentially connecting them) are all held within a 'CrappyWorld' object, which has an update loop (handling the physics), as well as other things such as methods to accept new bodies, a method to use an 'I_CrappilyDrawStuff' to render the current state of the world, and methods to mark bodies for removal, among other things. These methods are all thread-safe (albeit the main update method, with the methods to add bodies/connectors, use rather coarse-grained concurrency all sharing a single 'UPDATE_SYNC_OBJECT', but the drawing method uses more fine-grained concurrency with each collection of 'drawable' views of the bodies being synchronized individually).

The CrappyWorld itself, along with the update loop, has tuneable parameters; these are for gravity (constant force, as a vector, applied to all dynamic bodies, ignored by kinematic bodies), number of update iterations to perform in the update method, length of time to simulate in each update iteration (the 'deltaT' value), and number of euler sub-steps to use within each update iteration. The CrappyWorld class has static constant default values for these parameters; when creating an individual CrappyWorld, that individual world is given its own default values for these parameters, which a programmer can choose to specify themselves via the constructor, or, if they use the no-argument constructor, they're initialized to match the static defaults. Furthermore, in the update loop, there's the option to use the object's own default parameters for the update loop (via the no-arg update method), or specify all of them manually by the update method that takes parameters (which the no-arg method calls with the specified parameters).

The 'euler sub-steps' mentioned earlier is essentially a way of specifying a step size for integrating the euler method between each update iteration, except, instead of specifying it via step size, specifying it via how many steps are desired (with the total deltaT for the update iteration being divided by this to get the 'step size' for the individual substeps).

The update loop itself performs the following actions: it first attempts to see if there are any pending changes to the states of the bodies/connectors (usually from the game's main update loop) and resolves these changes (such as removing the bodies/connectors if appropriate), before then dividing forces/torque applied by the programmer by mass/moment of inertia. It will then try to find the bounds of the dynamic/kinematic bodies to set up the structure of the axis-aligned bounding box quadtree of them (explained further in the next segment). It then starts performing the individual update iterations. It first gets the connectors to apply the appropriate tension forces to the bodies they connect (based on distances, applied as a mid-timestep force), before then performing the first, preparatory euler sub-update, where it adds the constant gravity (multiplied by mass) to the centroids of each dynamic body, divides the constant forces for the timestep by mass/inertia, and then it adjusts the velocities, positions, angular velocities, and rotations for the bodies for this first substep. It then repeats the slightly simpler process of 'recalculate connector force -> perform euler substep' for the remaining substeps. It then finalizes the position/velocity changes from the euler substeps, before performing the collision handling (explained in the following section), then (after all of the collisions) calling the post-collision callbacks for all the bodies, handling any state changes to/pending removals of the bodies (again), and then repeating this process for the other update iterations. Finally, it will synchronize with each list of drawable bodies/the drawable connectors, and update them, concluding the update method.

The connector force is recalculated and reapplied each sub-update, as, even though it's arguably a bit inefficient to do this, it meant that the forces applied by the connectors would be a bit more accurate, which, at least in theory, means that the connectors are less likely to go haywire due to poor integration.

The default parameters within *CRAPPY* for the CrappyWorld specify a delay of 20 milliseconds per update, 100 iterations per update call, 2 euler substeps per update iteration, and an earth-like gravity of $\begin{bmatrix}0\\-9.81\end{bmatrix}$. *SpaceTow* uses a gravity of $\begin{bmatrix}0\\-0.95\end{bmatrix}$ instead, being close enough to the gravity of Thrust (calculated via manual timing and pixel measurements, see endnotesi) to not feel completely uncomfortable to play. The other parameters used for *SpaceTow*'s CrappyWorld are the same as the default parameters, as I observed that they worked well enough to produce something playable and believable as-is, meaning that further tuning was not a necessity.

Collision handling – Axis-Aligned Bounding Boxes and Quadtrees

I briefly mentioned this earlier, but *CRAPPY*'s collision handling logic has a 'broad phase' (relying on axis-aligned bounding boxes) and a 'narrow' phase. As mentioned earlier, every single rigidbody in *CRAPPY* should have a shape associated with it, and the shapes have axis-aligned bounding boxes (AABBs) around them, fully enclosing the shape's vertices (for edge shapes, however, the bounding box will attempt to enclose the full 'depth' of the edge, and, for edge clipping prevention purposes, the size of edge bounding boxes are incremented slightly (upper bound increased by (0.1, 0.1), lower bound decreased by (-0.1, -0.1)) and then enlarged by 10%, to make it a bit harder for a shape to clip past the edge area of the edge). The 'CrappyWorld' uses the AABBs within a quadtree data structure for 'broad phase' collision handling. Within the crappy/collisions package, there's the 'AABBQuadTreeTools.java' file, containing classes used for the static geometry quadtree, and the dynamic/kinematic quadtrees.

The static geometry quadtree is defined once, given to the CrappyWorld, and that will be the static geometry used by the CrappyWorld, whilst the dynamic/kinematic quadtree is recalculated (whilst being used) every collision frame. For ease of usability, the static geometry tree is supposed to be created via a factory method which takes a collection of static bodies as an input, returning the static geometry tree. The code creating the tree itself observes the bounding boxes of all 'candidate' bodies (which need to be divided up), tries to come up with an appropriate 'midpoint' to use to divide the AABBs into the appropriate child quadtrees, divides them up by seeing where the AABB's boundaries are in comparison to the selected midpoint (if the AABB straddles multiple subregions, that body will be considered to be in both/all of the regions it straddles), then, with these divided bodies, it tries again to divide them even further. If no bodies/only one body is present within a region, an empty/singleton leaf node is created for that region. Otherwise, depending on how successfully the current node divided the bodies, it may choose a different selection method for dividing the next few layers (maybe the median of upper bounds, median of lower bounds, or just the average from a random point in the bounds of each), however, if this is the third iteration in a row in the current branch of the tree which has not lead to a decrease in the number of individuals that it's currently attempting to divide, instead of wasting further space trying to divide them further, it'll just stop there and return a leaf node containing all of those individuals.

The kinematic/dynamic body quadtree is somewhat different. During the update loop, the world will try to get the overall bounds for all of the dynamic/kinematic bodies as they're being updated. The structure of this quadtree will have a depth of $\lceil \log_4 b \rceil$ (where b is the total count of dynamic and kinematic bodies), with all the nodes at a given depth having the same size as each other (using midpoints that evenly divide each segment into quarters), and this depth limit of logarithm base 4 was chosen to complement the tree's branching factor of 4, intended to limit the potential combinatorial explosion. After establishing this structure (with the structure being dictated before the first update iteration in the update method, with contents wiped at the start of each update iteration allowing it to be repopulated), the tree is finally used, and it's populated whilst collision detection takes place. Firstly, each kinematic body is given to the tree, and, based on the relationship between the body's bounding box and the midpoint of the current node, it'll attempt to find out which regions the body it possibly is in (using a depth-first approach). Upon reaching a leaf node, the body's bounding box is compared to that of all of the other bodies already there at that node, recording the bodies it collided with in a set, before this current body is added to the leaf itself. Eventually, these sets of bodies that had intersecting bounding boxes are combined together, along with the set of bodies with intersecting AABBs present in the static geometry tree (checked identically, via a depth-first strategy), and then collisions between the current body and all the other bodies that had an AABB overlap are handled (including the more in-depth checks, such as whether the bodies are allowed to collide, whether the actual shapes overlapped, etc). This is then repeated for the dynamic bodies (added to the tree which the kinematic bodies were added to, checking intersections en-route, and then combining those with static geometry intersections, before handling those collisions). Then, for the subsequent iterations of collision handling within the current update call, the kinematic/dynamic bodies are removed from the tree (the structure itself is not recalculated), allowing them to simply repopulate the tree with their updated positions.

Forces

CRAPPY is a mostly impulse-based physics engine; most forces on bodies are applied as impulses, factoring in the masses and moments of inertia of both bodies involved in a collision, applying the appropriate linear force (and torque) depending on where on the body the collision was.

For most collisions between moving objects, this works perfectly fine, however, I noticed that it wasn't working as intended for collisions between static bodies and non-static bodies. I started to get a bit concerned by these non-functional collisions, and eventually just decided to use a hacky workaround of simply overwriting the linear/angular velocities of the dynamic body (instead of applying the impulse which would have the same net result). When a dynamic body collides with a single static body, that body now behaves as one would expect, which is good. However, when there are two static bodies which have edges meeting at an acute angle, with both slanting downwards towards that meeting point (with normals pointing towards each other), the dynamic body would eventually fall into the area where it's colliding with both bodies, but, due to them overwriting the velocity instead of applying an impulse, it's possible for one of the static body to effectively push the body through the other static body, as it overwrites instead of merely negating the force applied by the other one. This could relatively easily be fixed by trying to calculate exactly how much force is needed to accurately have the same end result of the overwrite (without the issues of overwriting), but, due to time constraints, this will remain unpatched.

Similarly, the polygon-edge collisions have a few minor problems themselves. Initially, I tried perform these collisions via treating the polygon as a series of edge colliders as 'whiskers' between the vertices and the centroid (with the end-circles of these whiskers being at the vertices). This made sense, but it sometimes lead to some rather inconsistent behaviour where the polygon could sometimes clip through the edges, or, when 'resting' on the edge, rapidly jumping in place, with one vertex being pushed away causing another one to come back to the edge, which in turn got pushed away, and repeating over and over again, eventually flying away from the edge just as quickly as it already abandoned the concept of 'conservation of momentum'. Instead, polygons now attempt to collide with other bodies by effectively turning into a circle. Polygon shapes have a circle collider for their incircle (the biggest possible circle, with an origin at the polygon's centroid, contained entirely within the bounds of the polygon), as well as another similar collider, at the same origin, with an adjustable radius. When a polygon collision happens, it first attempts to collide the incircle of the polygon against the other shape, and if that fails, it attempts to work out the collision normal, and works out if the point at which it collides with the other shape is actually in bounds of this polygon; if it is, it then collides a circle of the appropriate radius against the other shape. I suspect that this may be the root cause of a certain bug which causes polygons to rest on edges by their centroids, not by the corners of the polygon (effectively clipping halfway into them). This could probably be fixed by applying some sort of constant 'penalty force' to keep the bodies an appropriate distance away from each other, basically letting them rest on each other (effectively simulating the 'normal contact force'), however, due to time constraints, that's unlikely to happen.

Connecting two bodies together

The 'CrappyConnector' connector in *CRAPPY* works via applying impulse forces. For the most part, it's nearly identical to the 'ElasticConnector' class supplied as part of the CE812 lab materials [4], but with a few minor improvements.

Firstly, it supports connecting bodies at arbitrary local positions to each other, instead of having to connect both at their centroids, and both bodies will receive the appropriate amount/lack of linear force and torque from the connector, considering where their local connection points are.

Secondly, it still uses Hooke's law to calculate tension, however, it supports a variety of approaches to truncate it (if a programmer so desires). They may truncate it via anything that implements the DoubleUnaryOperator functional interface, but there are several premade truncation rules which one can use instead (obtainable via a factory method, or by directly instantiating them). The simplest ones are the 'return zero', 'no truncation' and 'standard truncation' methods ('standard truncation' has a limit, and simply caps the input at ±limit). However, if smoother truncation, is desired they may opt for 'TanhTruncation' or 'PartialTanhTruncation'. 'TanhTruncation' simply returns $limit \times \tanh\frac{input}{limit}$, effectively ensuring all inputs will be on the hyperbolic tangent curve within range (-limit, limit). 'PartialTanhTruncation' instead defines a 'safe' proportion of the limit, only truncating anything outside of that limit, like so: $LET\ \boxed{sLim = [lim] \times p}\ AND\ \boxed{tLim = [lim] - sLim}$

$$IN\ \boxed{f(in, lim, p) = \begin{cases} \boxed{[in] \leq sLim} \xrightarrow{yields} in \\ \boxed{[in] > sLim} \xrightarrow{yields} \left(sLim + \left(\tanh\frac{[in] - sLim}{tLim} \times tLim\right)\right) \times sgn(in) \end{cases}}$$

The end result of this truncation method is that Hooke's Law will be used as-is whilst the result of it is within the 'safe' proportion of the limit, with only the excess above the 'safe' limit being normalized via tanh, but still ensuring that the truncation is gradual, and always within the limit, no matter how far outside the safe limit the raw value is. This truncation method was used for the ship/payload connector in *SpaceTow* (with a truncation limit of 150 and a 'safe' proportion of 0.75 (safe limit of 112.5), along with a spring constant of 50000 and a 'damping' level (to prevent oscillations) of 100).

Furthermore, there is a setting to let the CrappyConnector can define if the bodies it connects are able to collide with each other. This is achieved mostly via the CrappyBody class. Each CrappyBody has a randomly-initialized UUID[1], usable to quickly check if two bodies are identical without needing to deal with the full equals method, or many other variables. CrappyBodies also have a set of bodies they cannot collide with, in the form of a HashMap of UUIDs and 'IHaveIdentifier' interface objects (exposing only the UUID getter, with the UUID being the keys of the maps); when checking if a body is allowed to collide with another body (within the collision handler code), before doing the layer-based bitmask check (before any of the narrow-phase collision handling), it sees if the UUID of the other body is present in the 'cannot collide with' map of the current body; if present, the bodies are treated as not having collided. This 'cannot collide with' map in each body, along with a set of all connectors a body has (via the 'CrappyConnectorBodyInterface' interface) is updated whenever a connector to that body is made/removed. The connector itself calls a 'connector added'/'connector removed' method of the bodies ('added' called on construction, 'removed' called when the CrappyWorld disposes of the connector, but both methods pass the connector itself as a parameter), allowing the bodies to add/remove it from their set of connectors. The bodies check if the newly added/removed connector was one that had 'collisionsEnabled' set to false; if it was set to false, they synchronize the 'canCollideWith' map, clear it, and refill it with the other bodies it isn't allowed to collide with from the updated set of connectors. *SpaceTow* uses this functionality to ensure that, whilst the ship may crash into the payload before picking it up, there won't be any unwanted player frustration due to payload-related accidents whilst towing it.

Additionally, the connection forces are recalculated and applied during *CRAPPY*'s improved euler updates (before each substep), instead of merely being assumed to be constant throughout each improved euler loop. As noted in the comments for the update method for the provided BasicParticle class in the CE812 course material, acceleration during substeps of the improved euler method should, strictly speaking, not be constant for any distance-based forces, such as the tension force for connectors, meaning that the connector behaviour in that is incorrect. This problem has been addressed by *CRAPPY* and the CrappyConnector. During each substep iteration for each Euler update in *CRAPPY*'s update method, the connector applies forces to the bodies via their 'applyMidTimestepForce' methods, not the 'applyForce' methods. The 'mid timestep' version increments a 'mid-timestep' pending force vector/pending torque double in the CrappyBody class, instead of the ordinary pending force/torque variables; then, within the function handling the euler substep update, when adjusting the current velocity (after amending the position based on the current velocity), the appropriate forces from the ordinary pending force/torque variables are applied to the velocity, in addition to the mid-timestep force/torque (divided by mass/moment of inertia), before the mid-timestep force/torque is reset, allowing this non-constant, distance-dependent force to be accurately simulated. On the topic of the mid-timestep forces, I admit that there is some scope for the collision handling code (along with the relevant force calculations) to be done mid-timestep, instead of being done post-timestep. However, due to the potential overhead involved with that, I do not consider that to be worth doing.

Finally, the natural length of the connector may be declared explicitly (via the constructor which has a 'natural length' argument), or implicitly (via the other constructor). If a programmer opts to declare it implicitly, the natural length is set to be equal to the distance between the world positions of the connection positions on each body (and this

[1] Universally Unique Identifier. See RFC-4122 for an in-depth explanation: https://datatracker.ietf.org/doc/html/rfc4122. Java provides an implementation of RFC-4122.

distance is used as a fallback if the explicitly defined natural length is not finite). *SpaceTow* uses this implicit distance calculation when creating the towrope, as that prevents any unexpected pushing/pulling from happening due to the ship potentially not being at the exact correct distance for a predefined 'natural length'. The 'ICrappilyDrawStuff' interface also allows one to see, at a glance, how compressed/stretched a CrappyConnector currently is, by adjusting the hue of it proportionally to how compressed/stretched it currently is (being yellow when at it's natural length, turning redder as it gets more compressed (limit of pure red when it's been compressed to half its natural length), and greener as it gets stretched (with a limit at pure green after being stretched 1.5 times its natural length).

Bugs and unfinished features (which have not already been mentioned)

In the interests of everyone's sanity, I'm delivering this section via bullet points, to avoid wasting the reader's sanity on too much prose (issues not indented, discussion of causes/potential fixes in the sub-list underneath the appropriate point).

- The linear interpolation-based viewport scrolling (based on the distance between the visible midpoint and current ship position (or current payload position or the crash site of the ship/payload)) occasionally freaks out, causing the viewport to momentarily shift somewhere completely unexpected, usually when the player presses the 'any' button to respawn.
  - Part of the camera movement code involves some division and scaling of vectors, and I suspect that, on these rare occasions, an unexpected zero or non-finite value slips in somewhere there, or potentially from the code which obtains the 'lerp target' (the Vect2D which the viewport attempts to show) potentially returning an unanticipated invalid value when the ship is respawning. However, due to the inconsistency of this bug happening, it's rather difficult for me to draw a solid conclusion on it.
  - However, I have since implemented a fallback to limit the damage this causes when it does happen, by warping the camera back to focus on the ship's respawn position when it goes into the realms of infinity/NaN.
- Polygons rendered by I_CrappilyDrawStuff (such as the ship) occasionally 'flicker' in and out of being visible.
  - I suspect this might be due to the concurrent nature of the inner workings behind the I_CrappilyDrawStuff interface, and how the 'Timer' class included in Java Swing occasionally doesn't cooperate with concurrency.
  - Currently, the I_CrappilyDrawStuff interface accesses collections of 'drawable' objects in the CrappyWorld; each of these 'drawable' object collections are individually synchronized, and their contents are all updated (each collection in its own synchronized block) at the end of the CrappyWorld update method. Furthermore, when calling the 'drawable' interface methods for each CrappyShape/CrappyConnector, there's further per-object concurrency, preventing all access to the 'drawable' data whilst being read/modified by another thread. This means that, if the timings join up in just the wrong way, there could be a significant delay in the rendering of the world, which probably would cause the Graphics object to not bother trying to render anything particularly complex, such as any arbitrary 2D polygons.
  - Alternatively, the issue could be a very well-hidden bug in the default methods of the I_CrappilyDrawStuff and I_GraphicsTransform classes which, on occasion,

  could cause the world-to-screen transformation to return a stupid number. However, as such a bug would probably result in much more dramatic body deformations, affecting things besides the polygons, I suspect it's probably not that.
  - This could be fixed by trying to make the code responsible for calculating the drawable shapes a bit more time-efficient (taking less time to run, therefore causing less of a delay for the rendering), seeing if replacing the Timer in the GameRunner class with a SwingWorker helps it to not freak out at the concurrency, or perhaps making the drawable concurrency even more fine-grained (although trying to avoid ConcurrentModificationExceptions thrown by poorly-timed update and draw loops would still pose a problem).
  - Worst-case scenario, one could scrap the multithreading entirely, and instead run the update loop and draw operation in series. The game would probably slow down quite a lot, but, in theory, it should have plenty of time to render everything.
- Using 'I_CrappilyDrawStuff' to render the world.
  - The intent for that class was to be used as a bottom-of-the-barrel rendering solution, for ease of debugging/prototyping, and not to be relied on for rendering a finished game.
  - The time constraints for making a game after completely overengineering *CRAPPY* meant that I didn't even attempt to render things more elegantly (maybe even including things such as a thruster animation on the ship instead of relying on a terrible acapella thruster noise, proper sprites for the other game objects, etc).
- Kinematic bodies and line colliders are completely untested.
  - I couldn't work out a way of tastefully inserting these into the game within the time constraints, therefore, these have not actually been demonstrated. I'm not entirely sure if they work as-intended either, but I suspect that they won't do anything too unexpected.
- Lack of any proper 'normal force' (or things resembling that) from the collision shapes, as well as a lack of friction force to cause a proper 'roll' when a body looks like it should be 'rolling' down another one.
  - All collisions are applied as impulses, and there is no 'normal collision force'/'penalty force' when two bodies are resting on each other, making it somewhat easy for the bodies to get pushed through each other. Friction force also isn't a thing within *CRAPPY*.
  - A normal force would probably require some significant refactoring to implement correctly, as I would need to work out some mechanism for applying said normal force (which isn't really an impulse) to the bodies that are experiencing it.
  - A friction force could, in theory, be applied to circle shapes resting on lines/edges without needing a complete refactor, by, when a circle shape is moving against an edge in such a way that one would naturally expect it to roll, applying a force to do that. It could be a friction force vector applied on the point where the circle collides with the surface, pointing away from where the circle is going (to get it to roll that way), or just applying it directly as a torque (if desperate), but it's not entirely unfeasible.
- Lack of content in the game.
  - There are only three levels in the game, all of which are the same experience every time you play them, and there's no score/rating/other form of validation to encourage a player to spend more than five minutes playing the game.

    - I suspect that, had I not had to deal with time constraints, I would have been able to design at least a couple of additional levels (maybe including some kinematic bodies and lines in there), or throw in a rudimentary high-score system. However, the deliverable only has 3 levels, and that's the way it is.
    - Sure, there is the unexpected ergodic literature interlude in the third level, which could leave an impression on the player after they finish playing the game (if the terrible sound effects didn't already scare them off), but, at the end of the day, there's not much here.
- I don't know if the quadtree implementation used *CRAPPY* actually provides any real performance benefit compared to the naïve $O(n^2)$ collision-detection algorithm.
    - I haven't actually gotten around to properly assessing the performance of my quadtree implementations, so I honestly can't declare them to be efficient.
    - In a worst-case scenario (where there are a lot of bodies in the world, and all of them are concentrated in a rather small area, with quite a few overlapping AABBs), I suspect that my static and non-static quadtrees may have a time complexity which is worse than $O(N^2)$, potentially becoming something particularly abysmal like $O(N^3)$, especially if overlapping objects overlap at the boundaries of the quadtree regions (and especially if said overlap-at-boundary was at the midpoint of a particular region).
    - I suppose that this inefficiency could be mitigated somewhat if I were to rewrite the checking methods to use a more 'depth-first' approach to finding shape intersections. Currently, upon a shape reaching a leaf node, it performs a broad-phase (bounding box) check against every single object in there, returning the collisions from that leaf in a new set, and then each parent node on the way up tries to combine the leaf collision sets before passing that back to its parent, which does the same thing, until reaching the root node. This means that, when two shapes have overlapping AABBs and both aren't contained entirely within a single region, their broad-phase check is repeated multiple times, which is somewhat redundant as the result will be the same each time. If I were to rewrite it such that the root node created an empty set (or map, taking advantage of the UUIDs belonging to the bodies to provide a somewhat faster 'is present in' check than the normal hashCode method), and passing that down through the tree, with each leaf node modifying that set in-place (not bothering to check overlaps between shapes that have a collision recorded there, or perhaps I could take this idea further and propose recording known outcomes for collisions for all bodies encountered so far, ensuring no body has a redundant second check), that would, at worst, prevent the complexity from getting significantly worse than $O(N^2)$.
    - But, on the positive side, at least the quadtrees don't appear to give false positive/false negatives in the broad phase of collision detection.
- A minor memory leak (regarding the pooled M_Vect2D and M_Rot2D objects) when disposing of a CrappyBody.
    - For context, the M_Vect2D and M_Rot2D objects are pooled, via static SynchronizedQueues (with new instances of these classes only being constructed if one of these are needed and the queue is empty), and are supposed to be 'discarded' (leading to them being put back into the queue) when no longer needed. This is done as these are intended to be used whilst performing chained mathematical expressions involving vectors/rotations, as, if only the immutable Vect2D or Rot2D classes were being used, that would cause an astronomical

number of those immutable objects to need garbage collection during vector maths; whilst the M_ versions are designed to minimize the work that the garbage collector needs to do, by not giving it work.

- A few instance variables for a CrappyBody are in the form of M_ versions of those classes, being obtained from the pool as usual, however, when a CrappyBody is removed from the world (no longer being usable), these mutables don't get 'discarded' as they officially should be, instead being given to the garbage collector.
- I could easily fix that bug by, in the CrappyBody method dedicated entirely to the world discarding it, explicitly discarding those objects. However, I have decided that I won't, as this does, in some way, prevent the size of the mutable pool getting too out of hand.
- These objects are taken from the pool as usual when the CrappyBody is initialized, meaning that the size of the pool will decrease. If the size of the pool is ever zero, a new mutable is made. Most of the time, mutables are put back into the pool immediately after the mutable is needed, so, most of the time, there's usually just about enough mutables in the pool to get any common operations done with them done. However, as the CrappyBody's mutables aren't immediately put back in the pool, whilst they're away, the pool will increase in size to a natural size; but, if the CrappyBody's mutables were all put back into the pools, over time, that would cause the pools to get larger and larger, unless the rate of body creation could keep up with the rate of body discarding, and could eventually cause a massive memory leak, which the garbage collector won't be able to do anything about, due to the ridiculous surplus in the pools. So, to avoid that catastrophic issue, I'll let this apparent bug keep the population in check.
    - In simpler terms, it's like how trees and fossil fuels have stored carbon taken out of the atmosphere a rather long time ago, some carbon has since gone back into the atmosphere, but now because everyone's been burning the ancient carbon stores and re-releasing that carbon into the atmosphere everything's rapidly going downhill (but replace the carbon with the mutables, the atmosphere with the pools, the trees/fossil fuels with the bodies, and the burning with explicitly re-pooling the mutables when discarding of the bodies).
- Yes, I could just give the pools explicit size limits and leave it at that, however, the pools are ConcurrentLinkedQueue objects (non-blocking, thread-safe), used by all the threads at once, making it difficult to work out exactly how many are likely to be needed (especially if a developer using *CRAPPY* wishes to use some themselves), so I'm letting it naturally settle at a 'reasonable' size.

Reflection

I have mixed feelings about the outcome of this assignment. On the one hand, I have produced a somewhat functional physics engine, and a playable game which utilizes said physics engine. However, as you may have noticed from the previous section of this report, this deliverable has many shortcomings, and, even worse, it took considerably longer than I had predicted to produce it (which, to put it euphemistically, means that the next few days are going to be rather hectic).

In hindsight, I can see that many severe blunders were made during the development process of this game. Firstly, I delayed starting work on this assignment (although this delay was mostly due to having to work on a couple of assignments that were due in earlier, although I did spend a bit longer than what one may consider reasonable on those (even factoring in the medical disruptions during the term), so I suppose it probably ultimately is my fault), and then started work on the false assumption that I would be able to get most of it done within a couple of weeks, due to initially thinking that it would have been a relatively minor task to essentially port some of the enhancements I made to the provided physics engine during the lab sessions to a further enhanced physics engine. However, nearly 4 weeks later, and I'm only just finishing this off.

Then, when I started development, I made the fatal error of not using unit tests. At the time, I felt a bit reluctant to do that, as I wasn't entirely sure what exactly I expected the expected outcomes of the methods to be, and that would have posed problems for the validity of the aforementioned tests. For example, for most of the mathematical functions, the only way I could think of to work out what the expected outcome for those would be would have been running those methods (or writing out and running another implementation of them and running them), which could have been utterly redundant due to that ultimately testing a method to see if it has the same outcome as itself. Whilst that could be useful for regression tests (after it's known to work), that isn't of much use when writing the thing and getting it to work in the first place. Furthermore, for some parts of the engine (such as the static geometry quadtree), I'm still not entirely sure what the expected outputs of that are, even now, so I still don't feel able to perform regression tests on it. This meant that I didn't get around to the simple task of checking that things work until the early hours of the morning of the 2rd of January, after finally implementing the renderer, and then having to manually debug the many errors I found after that. The next time I work on anything like this, I need to write unit tests.

I will admit that, in context of the current situation I've found myself in, I've started to recognize *CRAPPY* as what it truly is: a product of hubris, with my current situation being merely the expected outcome of the Dunning-Kruger effect. Chances are that if I had not spent time overcomplicating it, maybe by omitting the polygons, which have many errors in their implementations anyway, or by not spending an entire week tunnel-visioning on a quadtree data structure which I didn't know how to test, it might have turned out a bit better in the long run. Or I could have just done the sensible thing, and simply use JBox2D instead, as that way I would have probably been able to have enough time to create a game which is actually half-decent, and to avoid the current predicament I am in regarding other work.

Finally, I highly doubt that anyone is likely to use *CRAPPY* in the future, for any reason (besides any sort of 'this is what not to do' cautionary tale). I already see it as a massive pile of regret and, for lack of a better word, the literal interpretation of its backronym name, so I don't think it's a good idea for me to attempt using it again. Furthermore, there's no reason for anyone else to want to use *CRAPPY*, due to the many shortcomings when compared to something like JBox2D.

To conclude, this project has, overall, been a rather large failure from start to finish, I am genuinely disappointed at my own blunders throughout the development process, and I must not permit myself to make those same blunders again.

Apportioning Credit and Prior Work

As mentioned earlier, whilst most of *CRAPPY* and *SpaceTow* were created from scratch, certain parts weren't. Here is a somewhat comprehensive list of everything that wasn't created from scratch, and who made them.

- *SpaceTow* was inspired by *Thrust*, by Jeremy Smith, 1986 [1].
- Classes inspired by other classes found in JBox2D/Box2D (by Daniel Murphy and Erin Catto respectively) [5] [3]
  - 'I_Rot2D', 'Rot2D' and 'M_Rot2D' in crappy.math
    - Heavily based on the 'Rot' class in JBox2D (and the original 'b2Rot' struct in Box2D). This class provided a rather elegant representation for a direction angle for a body (by representing an angle theta via 'sin(theta)' and 'cos(theta)', meaning that the values naturally wrap around instead of potentially getting larger and larger unless one manually caps them via modulo, and allowing one to easily rotate a vector by an angle due to pre-computing the cosine and sine values needed by the rotation matrix), so I figured that there wasn't much harm in using the same approach myself.
  - CrappyWorld, CrappyBody, A_CrappyShape, etc.
    - Inspired by the concepts behind the 'world', 'body', and shape 'fixture' classes
  - I_CrappilyDrawStuff
    - Based on the principle behind the DebugDraw/b2DebugDraw classes.
  - CrappyConnector
    - Several of the enhancements I made to the original ElasticConnector to produce the CrappyConnector were based on some joint functionality (such as arbitrary local positions, disabling collisions between connected bodies, etc) offered by JBox2D/Box2D's joints.
- Classes based on course material provided by Dr. Michael Fairbank for the 2021-2022 CE812 Physics-Based Games module at the University of Essex [4]
  - 'I_Vect2D', 'Vect2D', 'M_Vect2D'
    - Heavily based on the Vect2D class provided as part of the course material.
  - CrappyConnector
    - Heavily based on the ElasticConnector class provided in the course material
  - The game of *SpaceTow*
    - It's ultimately an extension of the fourth lab task given in that module (albeit in a different engine).
  - A lot of the collision handling code, the update loop, along with the theory behind *CRAPPY* itself
    - Based on concepts taught during this course
  - crappy.graphics
    - Somewhat based on the rendering code within the BasicPhysicsEngine implementations provided in the course material
- Vect2DMath and CrappyCollisionMath classes implement theory from several sources, these being:
  - Dr. Michael Fairbank (theory explained during the CE812 Physics-Based Games module at the University of Essex) [4].
  - Jeff Thompson's website/book on the topic of collision detection [6].
  - Piyush Rajendra Chaudhari's implementation of the shoelace algorithm for working out the area of an arbitrary polygon [7].

        - Paul Bourke's very comprehensive notes on geometry [8].
        - Martin Thoma's explanation of an algorithm for checking if two lines intersect [9]
- The contents of the 'crappy.utils.lazyFinal' package
    - These classes are adapted from Tetyana Kolodyazhna's implementation of a 'write-once-read-many' class [10].
- Prior work of my own which was reused in *SpaceTow* and CRAPPY
    - crappyGame.misc.AttributeString.java
        - The latest (and least terrible) iteration of a class I originally created for *Epic Gamer Moment – The Game* in November 2019.
    - Most of the 'crappyGame' package
        - Architecture and several of the utility classes are based on architecture/utility classes which I have used for several prior projects between 2020-2021
            - Some of which was itself adapted from sample code provided by Professor Dimitri Ognibene as part of the CE218 Computer Games Programming module at the University of Essex [11], but most of this sample code has been significantly refactored.
    - The background images for the levels
        - Initially produced as skybox textures for *Inconvenient Space Rocks 2*, made between May/June 2021.
    - The sound effects
        - The sound effects were previously improvised by me between 2020-2021 during the development cycles of *Inconvenient Space Rocks*, *Muffin Mania*, *Inconvenient Space Rocks 2*, and *The Button Factory*.
        - The background music, however, was improvised, by me, specifically for *SpaceTow*. The main game theme does take some inspiration from *Deep Purple – Space Truckin'*, whilst the 'conversational' theme is inspired by *Valve Studio Orchestra – Upgrade Station*, *Herp Albert and the Tijuana Brass – Spanish Flea*, and *Kevin Macleod – Local Elevator*, but the 'an ending' theme was an entirely original improvisation.
- The classes in 'crappy.utils.containers', along with 'crappy.utils.ArrayIterator' probably bear a very striking resemblance to many existing implementations of these abstract data types, but I made those implementations of those helper classes myself.

Bibliography

[1] J. Smith, *Thrust,* Superior Software, 1986.

[2] iforce2d, "Using debug draw - Box2D tutorials - iforce2d," 14 July 2013. [Online]. Available: http://www.iforce2d.net/b2dtut/debug-draw. [Accessed 31 December 2021].

[3] E. Catto, "Box2D: Overview," 2021. [Online]. Available: https://box2d.org/documentation/. [Accessed 12 November 2021].

[4] M. Fairbank, *CE812 Physics-Based Games,* Colchester: University of Essex, 2021-2022, autumn term.

[5] D. Murphy, "JBox2D: A Java Physics Engine," 2014. [Online]. Available: http://www.jbox2d.org/. [Accessed 12 November 2021].

[6] J. Thompson, "Collision Detection," 10 October 2020. [Online]. Available: https://jeffreythompson.org/collision-detection/index.php. [Accessed 24 December 2021].

[7] P. R. Chaudhari, "Area of Polygon: Shoelace formula," OpenGenus Foundation, [Online]. Available: https://iq.opengenus.org/area-of-polygon-shoelace/. [Accessed 2021 December 26].

[8] P. Bourke, "Geometry, Surfaces, Curves, Polyhedra," [Online]. Available: http://paulbourke.net/geometry/. [Accessed 2021 December 26].

[9] M. Thoma, "How to check if two line segments intersect · Martin Thoma," 21 February 2013. [Online]. Available: https://martin-thoma.com/how-to-check-if-two-line-segments-intersect/. [Accessed 2021 December 28].

[10] T. Kolodyazhna, "java - Declare final variable, but set later - Stack Overflow," 2016 July 10. [Online]. Available: https://stackoverflow.com/questions/11583502/declare-final-variable-but-set-later/38290652#38290652. [Accessed 2022 1 1].

[11] D. Ognibene, *CE218 Computer Games Programming,* Colchester: University of Essex, 2019-2020, Spring Term.

---

[i] I observed that the spaceship in *Thrust* falls a distance of ~190px from 0 velocity in ~2.5 seconds (on a world with a visible height of ~650px, with these measurements obtained by screenshotting *Thrust* as displayed by the emulator on bbcmicro.co.uk). Applying some physics equations gave an acceleration due to gravity of 60.8px/s^2, then factoring in the screen size, and an arbitrary 'visible world' size of 10, I got a world gravity of ~-0.93, which I rounded to -0.95 because -0.95 seemed to be a less awkward value, also considering the inaccuracies in my measuring methodology.

$$displacement = \left(\frac{initial\ velocity + final\ vel}{2}\right) \times time \equiv 190 = \left(\frac{0+v}{2}\right) \times 2.5 \equiv \frac{190}{2.5} = 76 = \frac{v}{2} \equiv \boldsymbol{v = 152}$$

$$final\ vel = initial\ vel + (acc \times time) \equiv 152 = (a \times 2.5) \equiv \frac{152}{2.5} = a = 60.8\ gravity\ screen\ scale$$

$$world\ grav = \frac{pixel\ grav}{visible\ pixels} \times visible\ world \equiv \frac{60.8}{650} \times 10 = 0.93\dot{5}3846\dot{1}\ world\ gravity$$