Meganck Jelte

Custom 2D Physics Library

Graduation work 2016-17

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

The aim of this paper is the creation of a custom cross-platform 2D physics library. It will focus on the in-depth creation and underlying algorithms used, as well as the systems and structures, which will be used to make the library able to handle scenes with hundreds of objects, which isn't uncommon in medium to large scale games, while only trying using a couple milliseconds of each frame's update time, so not to impact the time the game logic has to update. The library will need to be simple enough, so that most people can understand it, while powerful enough to handle the simulations.

INTRODUCTION

The paper’s purpose is to create a cross-platform 2D physics lib, which can be linked as a static library to any external project. The library will need to handle a range of different functions: Forces, Collisions, Joints, Constraints, …

The challenge of this project does not only come from implementing these systems, but also to optimize them, since the library is expected to have a limited timeframe of a couple milliseconds to update small (objects) to large (hundreds of object) simulations.

To be able to do these simulation in a limited timeframe, optimizing the project is very important, there are a lot of possible ways, which can be combined, to handle this. These include different data structures, systems and algorithms.

There are also problems that can influence the accuracy of the simulation, some of which are caused by optimizations, while other are because of the inner working of a physics library (floating point errors, timesteps, …)

The library also needs to be determent, which means that, when a simulations is started multiple times with the exact same parameters, it should result in the exact same result, so random values cannot be used in the library.

RESEARCH

While most research is done during implementation, since a lot of problems can arise during its implementation, while the case study expands on the info in the research, the research will be separate of the case study, but it will try to follow the same order as the research as close as possible, to make it easier to comprehend.

1. ALLOCATORS

A major slowdown of any project is its allocations. When using C++’ *new* and *delete* keywords, the underlying logic uses 2 C functions: *malloc()* and *free()*, which, because of their inner workings, can slow the program down a lot. The reason for this is that, to allocate or deallocate/free memory, needs to use functionality which normally isn’t available to the program.

The standard way of memory allocation: *malloc()* has some inherent problems, the allocator can cause memory fragmentation, which wastes memory and can cause performance degradation. Another reason for *malloc()*’s poor performance, is that it needs to be general, able to allocate memory of any size, on any system, and in every possible instance (including multi-threaded applications).

A solution to this problem is custom build allocators, which are less general, but are faster than *malloc()*. The library uses 2 types of allocator. The following allocators are based upon the allocators present in Box2D.

1.1. STACK ALLOCATOR

The stack allocator uses a LIFO (Last In First Out) structure to handle memory, based on the stack container. Which means that the memory which is allocated last, needs to be deallocated first. Internally it has a fixed size memory block and a limited amount of entries, which limits the amount of allocation possible, but speeds up the allocation and deallocation. If not enough memory is available, the allocator uses the default allocator.

1.2. BLOCK ALLOCATOR

The block allocator uses fixed size blocks of memory to allocate. This allows the allocator to quickly allocate memory. When allocating, the allocator finds the best fitting block size to fit the data, after which it can either return a pointer to an already existing, but empty block, or create a new block in the reserved memory. Unlike the stack allocator, the block allocators can allocate new chunks of memory whenever the allocator when the allocators chunks are full. The size of the allocation is limited by the size of the biggest block, when the allocation needs more data, the default allocator is used.

2. PHYSICAL PROPERTIES OF A BODY

A physics body is a collection of properties and shapes, that represent an object. Some of these properties will be explained in detail, further on in this paper.

The object has the following properties:

- Collection of shapes
- Position and angle
- Linear and angular velocity
- Force and torque
- Mass
- Center of mass
- Inertia

2.1. SHAPES

One of the most important parts of the body are the shapes, which represent the parts of the body that interact with the world, without these, collision could not happen.

Shapes can come in multiple different forms, each needing its own ways of calculating properties and collisions. These possible shapes are: circles, edges, chains (collection of edges) and convex polygons.

The shape itself has properties, some of these will be expanded upon later:

- Relative position compared to the body
- Friction
- Restitution
- Density
- Mass
- Center of gravity
- Inertia

2.2. FRICTION

There are 2 types of friction: static and dynamic. Both types of friction are defined by a scalar value between 0 and 1. 0 meaning no friction, so a surface won't give any resistance to an object and 1 meaning that the object is extremely hard to move, because of the large resistance.

2.2.1. STATIC FRICTION

Static friction simulates the resistance created by the interaction of 2 surfaces, where a higher value refers to an high roughness of an object. It is always applied to an object and is dependent on the normal force, which is the force pushing back on the body from the surface it is against, which itself depends on the following formula:

$$N = m * g * \cos(\theta)$$

Where: N is the normal force1

m is the mass of the object

g is the acceleration caused by gravity

θ is the angle of the surface a body is resting on, compared to the perpendicular to gravity

Using the normal force, static friction is defined with the following formula:

$$F_s = \mu_s * F_N$$

Where: F_s is the force of static friction

μ_s is the coefficient of static friction

F_N is the normal force

2.2.2. DYNAMIC FRICTION

Dynamic friction, also known as sliding friction, is like static friction, but in contrast to static friction, dynamic friction is only applied when 2 surfaces are sliding over each other. The coefficient of dynamic friction is normally a bit smaller than the static friction, but can easily be the same value.

Dynamic friction is defined by the following formula:

$$F_k = \mu_k * F_N$$

Where: F_k is the force of dynamic friction

μ_k is the coefficient of dynamic friction

F_N is the normal force

2.3. RESTITUTION

Restitution can also be called the 'bounciness' of an object, is the ratio between the incoming and outgoing velocities. This value decides the velocity that an object will have after a collision. This value is a scalar between 0 and 1, 0 means that the collision is completely inelastic, which means that all velocity in the direction of the collision of the hit will be canceled out. While 1 means that the collision is completely elastic, this means that all incoming velocity will be reversed and the object will bounce back and the velocity will just be in the opposite direction.

2.4. DENSITY, MASS AND CENTER OF MASS

Density and mass are dependent on each other, while in 3D, the mass is the product of the density and volume, in 2D, this is the product of the density and area of a shape. The mass of the object has a large influence on how this object will behave, including forces, that are applied on and by it, the friction applied on the object and the inertia of the object.

The object does not only have a mass, but also a center of mass or the point where the mass equals out from all directions of the object. This point has influence on the torque applied on an object, when it is hit off-center.

2.5. ANGULAR INERTIA

The angular inertia, also referred to as just 'inertia', of an object is also known as its rotational mass. Inertia influence the torque applied to an object, or how easily the object will start spinning, and object with a high inertia needs more force to be rotated and will rotate slower, compared to an object with a relatively lower inertia.

3. FORCES

Forces can be split up in 2 ways: the duration in which the force is applied and on which component of the velocity the force is applied. A force can be applied on 2 components of velocity:

3.1. LINEAR FORCE

Linear force, also referred to as just 'force', is applied on the linear component of the velocity. This force is the most well know, described by the famous formula: $F = m * a$, but instead being expressed in 2d space (with vectors). A force is applied over time and therefore is dependent on the time, combined with the mass of an object, dictates the change in linear velocity. Force can just be accumulated and then applied to the object.

A linear force can also be applied almost instantly on an object, called a linear impulse. Because this property, the it can be applied without needing time in its formula, but is instead defined as a change in velocity, as defined with following formula:

$$F = m * \Delta v \ or \ F = m * (v_1 - v_0)$$

Where: F is the applied

m is the mass of the body

Δv is the difference in velocity

v_0 is the velocity before the force is applied

v_1 is the velocity after the force is applied

3.2. ANGULAR OR ROTATIONAL FORCE

Angular or rotational force, also referred to as ‘torque’, is comparable to linear force, with 2 main distinctions: The force is applied to the rotational component of velocity and angular inertia being used instead of mass. Like linear force, torque is applied over time, which with the inertia, dictates the change in the angular velocity.

An angular force can also be applied instantly on an object, called angular impulse. Like linear impulse, the impulse can be applied without the need of time in the formula, but can be defined as the change in angular velocity, as defined by the following formula:

$$\tau = I * \Delta\omega \text{ or } \tau = I * (\omega_1 - \omega_0)$$

4. INTEGRATING VELOCITY AND POSITION

A physics engine needs to be able to handle the velocity and position of an object. These forces are integrated in 2 steps, to be able to handle collision, explained further on in this paper.

4.1. INTEGRATING VELOCITY

Velocity integration is the more complex of the 2 steps, since multiple factors need to be kept in mind. One of these factors is the gravity working on a specific object, the gravity is integrated using the following formula:

Gravity: $\overrightarrow{v'} = \vec{v} + \vec{a} * t$

Where: $\overrightarrow{v'}$ is the new velocity

$\vec{v}$ is the original velocity

$\vec{a}$ is the acceleration due to gravity

t is the time that has passed since the previous step

The next step is to integrate the forces upon the object, using following formulas:

Linear velocity: $\overrightarrow{v'} = \vec{v} + \frac{\vec{F}}{m} * t$

Where: $\overrightarrow{v'}$ is the new linear velocity

$\vec{v}$ is the original linear velocity

$\vec{F}$ is the force applied to the object

m is the mass of the object

t is the time that has passed since the previous step

Angular velocity: $\overrightarrow{\omega'} = \vec{\omega} + \frac{\vec{\tau}}{I} * t$

Where: $\overrightarrow{\omega'}$ is the new angular velocity

$\vec{\omega}$ is the original angular velocity

$\vec{\tau}$ is the torque applied to the object

I is the inertia of the object

t is the time that has passed since the previous step

4.2. INTEGRATING POSITION

This is the simplest of the 2 steps, but can be done in multiple ways, the physics engine uses a method called Euler integration. Euler integration is done by using the calculated velocity from during the current timestep. The integration is provided by the following formulas:

Position: $\vec{P'} = \vec{P} + \vec{v} * t$

Where: $\vec{P'}$ is the new position

$\vec{P}$ is the original position

$\vec{v}$ is the linear velocity

t is the time that has passed since the previous step

Rotation: $\theta' = \theta + \omega * t$

Where: θ' is the new angle

θ is the original angle

ω is the angular velocity

t is the time that has passed since the previous step

5. COLLISION

Collision is one of the most important tasks of a physics library, but also the most intensive, therefore the collision stage of the update needs to be heavily optimized.

5.1. COLLISION DETECTION

The first step to update collision is to detect if a collision is happening, since a lot of object can be in a scene, this phase of the physics update can contribute a lot to the needed update time. To optimize the detection, it is split up into 2 phases, the broadphase and narrowphase.

5.1.1. BROADPHASE

The broadphase needs to iterate, in an efficient way, throughout the world and find shapes which have a possibility to collide. This can be done using AABBs (Axis Aligned Bounding Boxes), OBB (Oriented Bounding Boxes) or circles that encapsulate the shape. In this paper, AABB will be chosen, because of their simplicity.

Each shapes AABB is also padded using a small value, which can give more possible overlaps, but lowers the chance of errors, since all shapes, except the circle, have a small ‘skin’. (An example is an edge aligned to an axis, without the padding, the chance that the collision will be properly updated, will be small).

An AABB contains the outer most values of a shape on the x and y axis. So, to compare if 2 AABBs are overlapping 4 simple comparisons can be performed. This phase can be done in multiple ways, the simplest, but slowest manner to do this is to go over all shapes in a scene and compare their AABB to those of all other shapes. But this can also be done by using specific data structures to speed up this phase, like an AABB tree, quadtree, ... (For more info on the way used in this paper, please check the case study).

For each pair of overlapping AABBs, the broadphase creates a contact from the AABBs' parent shapes. A contact does not mean that the shapes overlap, but means that there is the possibility of an overlap.

5.1.2. NARROWPHASE

The narrowphase takes in the contacts created by the broadphase and checks if the shapes overlap each other. Each pair of shape types, need their own algorithm, since the collision between a 2 sphere is different than the collision between 2 polygons. These algorithms produce a contact manifold with information that will be used to calculate the collision response.

During the algorithms, there will be referred to the skin thickness, this value works as a small tolerance during the checks, since without this, floating points errors could give the incorrect result back, when executing the algorithm.

The collision manifold is created during this step, info about the creation is in the case study.

The different algorithms and used of the collision detection and purpose of the manifold will be explained is a simplified manner:

5.1.2.1. CIRCLE-CIRCLE COLLISION DETECTION

Collision between 2 circles is the simplest algorithm. The algorithm works by comparing the distance between the center of the 2 circles to the sum of the circles radii.

5.1.2.2. EDGE-CIRCLE COLLISION DETECTION

This algorithm works by first projecting the circles center on the edge, which will result in the closest point on the edge. The location on the edge can be easily calculated using the dot product of the edge's direction and the vector from the first outer most point of the edge to the center of the circle. If this value would lie outside of the edge, one of the outer points will be used.

After calculating the closest point, the distance between these can be easily compared to the sum of the circles radius and the skin thickness of the edge.

5.1.2.3. POLYGON-CIRCLE COLLISION DETECTION

This algorithm is an extension of the edge-circle algorithm, as it uses the same calculation, but goes over each side of the polygon and.

5.1.2.4. EDGE-POLYGON COLLISION DETECTION

This algorithm uses the principle of SAT (Separating Axis Theorem) (more info can be found in the next algorithm). First, the algorithm goes through all points and calculate if they are above or below the line, if all points are either above or below the line, it means that there is no collision.

5.1.2.5. POLYGON-POLYGON COLLISION DETECTION

This algorithm also uses SAT (Separating Axis Theorem, also called Hyperplane Separation Theorem). The theorem proclaims, that for 2 convex shapes, if there is any possible axis, on which you can project all points of both shapes, and those regions where the points are located, don't overlap, that the 2 shapes are disjointed, or are not overlapping.

The algorithm goes through each edge of shape 1 and project all points of shape 2 on to it and stores all points below the line. After the first time, it only iterates through the points that are possibly in the shape (the points that are below all previously checked lines.

After this finished, the algorithm does the same projections again, but this time with the points of shape 2 projected on shape 1. If any points are in a polygon, this means that the shapes are colliding.

5.1.2.6. OTHER SHAPE COMBINATIONS

All other possible combination of shapes is not calculated, since any collisions made up out of only edges or chains don't collide, since these shapes can only be added to static shapes, which can't interact with each other.

5.1.2.7. CONTACT MANIFOLD

The contact manifold is a container of data used in the collision response, this includes the points needed in the calculation, the normal for the collision and the penetration of the 2 shapes. More info about the creation of the manifold can be found in the case study.

5.2. COLLISION RESPONSE

The collision response uses the data created by the collision detection to try and recreate the most realistic response there is. But since the step-wise update of a collision algorithm, this will still have issues that are very hard to solve. The collision response needs to solve 2 properties: velocity and position.

5.2.1. VELOCITY RESPONSE

The first step of the collision response is the velocity response. This can be done in 2 simple steps: restitution and friction.

The first step uses the coefficient of restitution and gives the force which both bodies will experience on at the point of collision. First the normal speed needs to be known. This can be done by calculating the velocity at the points where the collision happens, using the bodies linear and angular velocity. Using those the relative force can be calculated and using the dot product with the normal. The resulting velocity can then be calculated using the coefficient of restitution. Since this happens instantaneous, the resulting velocity can be used as an impulse on both objects. This force is than scale proportionally dependent on the on the mass of each object.

After this, the second step can be calculated, using the previous velocities and the velocities updated in the previous step, we can calculate the tangent speed of the objects, which can then be used, together with the friction to calculate the addition impulse given in the orientation of the tangent. Which depending on whether, using the tangent speed, the force can win against the resistance of the static friction, will give a different response, by changing the velocity at collision points accordingly.

5.2.2. POSITION RESPONSE

The system used for position response is simplistic, it moves both shapes depending on the displacement. If both bodies are dynamic, each gets moved by half of the displacement along the normal, if one body is not dynamic, the dynamic body gets moved by the full displacement along the normal.

6. CONSTRAINTS AND JOINTS

Constraints and joints are ways to limit the movement of objects. A constraint is a general way to limit the movement of an object in space, while a joint is more specific then a constraint and can be used to both limit movement of 2 bodies compared to each other or a body in space.

6.1. CONSTRAINTS

The constraint can limit movement in 2 different ways. The constraint can limit the angle of a body, which therefore also cancels out the angular velocity when a limit is hit. But it can also limit the movement is space, the movement is fixed on a certain axis in space, with a tolerance, which works perpendicular on the axis, which can also have a limit on how much an object can move on that axis.

6.2. JOINTS

There are currently 4 types of joints: Fixed, Distance, Revolute and Prismatic. The joints are connected at a point, relative for each body, but the joint can also connect a body to a position in space. These will all be given a simple explanation, with the expectation that the joint would connect 2 bodies.

6.2.1. FIXED JOINT

The fixed joint fixes the orientation and position of each body at a given point, this results in the movement of 1 object affecting the other object.

6.2.2. DISTANCE JOINT

The distance joint makes sure that the given points always keep the same distance between them.

6.2.3. REVOLUTE JOINT

The revolute joint is a more advanced joint. It makes sure that the bodies rotate around the given points, which will be brought to the same position. In addition to this, the joint can also limit the rotation relative to both bodies and can apply an additional angular velocity on both bodies.

6.2.4. PRISMATIC JOINT

The prismatic joint has the same movement restriction as a hydraulic cylinder. It limit the movement of the point on the second body to an axis, relative to the point on the first axis.

CASE STUDY

1. ALLOCATORS

The allocators are based upon the allocators included with Box2D, but they are made to fit the implementation of the library.

1.1. STACK ALLOCATOR

The stack allocator has a predetermined size to allocate and is made to be used as a frame allocator, which means it can not only allocate and deallocate, but also reset the data at the end of each update.

When allocating new data, a new entry is created in the allocator, it includes the size, a pointer to the data and if the default allocator was used. These are kept in the same order as the allocations happened, which will be used to check the order of deallocations. When deallocating, the data should be the same as the data stored in the last entry and it will then make the used memory free for new allocations.

The allocator can also be reset, when this happens, the allocator can use the list of allocation to revert the allocator to its original form.

1.2. BLOCK ALLOCATOR

The block allocator uses an array of predetermined block sizes, which will be compared to the size needed when allocating.

When allocating, the allocator finds the best fitting block size, if an empty block with the same size is available, it will use the first available block, otherwise it will create a new block. When deallocating, the allocator will just add the block to a list of free blocks.

The block allocator has no reset, since it will be used during the duration of the program, instead for only one update.

2. WORLD, BODY AND SHAPE

2.1. WORLD

The world is the class which updates all physics. The world contains most systems needed to handle the physics: allocators, list of bodies, constraints, joints, …

The world is also the main interface to interact with. It allows the user to create bodies, shapes, constraints and joints, update the world by giving in a variable time delta or step through the physics with a certain timestep. It also lets the user set the gravity of the world and callbacks.

2.2. BODY

The body contains all properties which are independent to the shape (as explained in Research 1.). The body is also a node in a linked list, having members pointing to the next and previous body that exists. It also holds a list of contact and joint nodes, which allows it to interact with the contact and joints its attached to.

The body is the object where you apply to, both linear and angular, this is one of the only ways to interact with the body, since allowing someone to move the body manually could cause problems in the physics simulation.

To create a body, a body definition is created, which includes the initial data of the body, including start position.

2.3. SHAPE

A shape can be 1 of 4 types: Circle, Edge, Chain and Polygon. These have the default properties of a shape, (as explained in the research 1.2.). Each shape also links to the other shapes in its parent body. While most parameters of a shape are not adjustable after creation, the user can still change the material.

The shape has 2 steps to add it to the scene, first the shape is created using a definition by the world and is then added to its parent body.

3. UPDATING PHYSICS

Physics are update in a specific order, to try to reduce the inaccuracies during the update. The steps will be explained below, and some will have a more in-depth explanation after this.

1) Velocities are integrated (as explained in Research 4.1.)
2) Positions are integrated (as explained in Research 4.2.)
3) Constraints are solved
4) Properties, which collision relies on, are prepared
5) Collision broadphase is updated
6) Collision narrowphase is updated
7) Collision's velocity response is updated
8) Collision's position response is updated
9) Joints are solved
10) AABBs are update for user to use
11) Velocity is limited
12) Body sleep is updated

4. COLLISION

The collision phase will be expanded upon the knowledge given during research (as explained in Research 5.)

Collisions also call events, which the user can use, each event gives the contact, which can be used in any way. There are 5 events: OnContactCreate, OnCollisionEnter, OnCollisionStay, OnCollisionLeave and OnContactDestroy.

4.1. COLLISION DETECTION

4.1.1. BROADPHASE

Since the broadphase's speed is directly dependent on the amount of bodies in the scene, it needs to be optimized. While it is possible to go through each shape and compare them to every other shape, this would be very time consuming, therefore a data structure is used, called an AABB tree. The implementation is based upon the implementation of box2d's b2DynamicTree.

An AABB tree is a binary tree where the 'key' is represented by an AABB. But since unlike a normal tree, the 'key' of a parent node is the combined AABB of the child nodes, any node other than a leaf node (end node) can't store data to an object, causes the tree to use more memory and needs to be implemented differently. (Basic knowledge of binary trees can help make the following info easier to understand.)

Adding or subtracting a node from the tree or even updating the AABB of a leaf, means that the tree needs to be rebalanced after each operation.

When adding a node, the tree needs to traverse the tree and find the most suitable node, which here is the node with the lowest cost, for which the cost is calculated based upon the size an AABB should increase, after which the tree is balanced.

Removing a node is simply done by freeing it and letting the sibling take over the place of the parent, after which the tree is balanced.

Balancing the tree compares the depth of 2 nodes, where after nodes are possibly rotated. When rotation is done, the parent's AABB becomes the combined AABB of the child nodes.

To find overlapping shapes in the tree, all nodes that have an AABB that overlaps the given AABB are checked, if the node has children, these are added to a stack to be searched, if not, the id of the node will be given to a callback, which can use the id to get the data associated with the node.

4.1.2. NARROWPHASE

The narrow phase uses extension of the simple algorithms explained in the research (as explained in Research 5.1.2.). The algorithms use the data is creates to check whether 2 shapes are colliding to create a contact manifold. The contact manifold includes contact pairs, which has the data for the contact, any contact with a circle has 1 pair and 1 or 2 contacts for a polygon, more contacts can be created by a collision with a chain.

Each pair contains 2 points, these are points on each object, which approximate where the 2 shapes would have hit, based on the normal, which is also included in each pair. These 2 points, created during the collision test, represent the closest points on the shapes.

The contact manifold used here is not only possibility to create a contact manifold, but the one best suited to the system used.

4.2. COLLISION RESPONSE

The collision response has mostly been explained in research (as explained in Research 5.2.).

During this phase, the library goes over al touching contacts (contact with a filled in manifold) and processes the contact. Each force that is applied, it is relative the amount of contact pairs, so when a contact manifold contains multiple pairs, the forces applied on the object will be divided by the amount of pairs, to limit unexpected behavior.

5. CONSTRAINTS AND JOINTS

5.1. CONSTRAINTS

Constraints are updated before collision, to prevent shapes colliding when they should be able to do it. When the rotation is limited, the angular velocity is canceled out to prevent problems cause by it during collision, e.g. when a circle spins and there is friction, it will start moving, so if the angular moment isn't limited, the circle will look like the sphere is moving too fast on a surface with high friction.

When the linear movement is limited, the linear velocity is adjusted so that the collision will act correctly, without this, the body can act in a weird manner.

5.2. JOINTS

Joints are updated after collision, if not, it can result in unwanted behavior. This could give issues with high velocities and restitution, but implementing it in a different way, so it would behave more correctly, would take up significantly more processing time, which could limit the application that uses the library.

Each joint also is connected to the body, in this way the user can access the joint from the bodies it is connected to.

Another problem which can be caused by joints, it that the joint can cause bodies to intersect, so the joint impacts whether 2 shapes can collide.

CONCLUSION

REPEAT THE MAIN TOPICS, DISCUSS YOUR MAIN FINDINGS, DISCUSS THE END RESULT.

The physics library is a challenging project, while the formulas used can be surprisingly simple, the implementation can be hard.

The collision is the hardest part to make and to make sure that it works properly, but because of the incredible amount of different cases a collision can happen, even if everything seems to work correctly in 1 situation, it can make it that another situation doesn't work at all, trying to fix these issues can take a lot of time.

Constraints and joints can complicate the calculations even more, since they can cause unexpected problem when combined with each other and/or with collisions.

At the end the project worked quite well, but a physics library is definitively a project which needs more than 6 weeks to reach a stable phase, a good example for this is Box2D, of which the development started in 2006 and is still in active development, while still having issues and bugs discovered.

REFERENCES

C dynamic memory allocation (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/C_dynamic_memory_allocation

Fragmentation (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Fragmentation_(computing)

Erin Catto (n.d.) Box2D source code, Github. Retrieved from https://github.com/erincatto/Box2D

Impulse (physics) (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Impulse_(physics)

Friction (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Friction

Coefficient of restitution (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Coefficient_of_restitution

Moment of inertia (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Moment_of_inertia

Hyperplane Separation Theorem (n.d), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Hyperplane_separation_theorem

Randy Paul (2013) How to create a custom 2D physics engine: The basics and impulse resolution, Tutsplus. Retrieved from https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331

Binary tree (n.d.), Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Binary_tree

APPENDICES