Module Interface Specification for Chipmunk2D Game Physics Library

Luthfi Mawarid

August 19, 2016

Contents

1 Introduction

The following document details the Module Interface Specifications for the implemented modules in the Chipmunk2D Game Physics Library. It is intended to ease navigation through the program for design and maintenance purposes. Complementary documents include the System Requirement Specifications and Module Guide.

2 Notation

Chipmunk2D uses six primitive data types: Booleans, characters, double-precision floating-point numbers (doubles), as well as signed and unsigned integers. These data types are summarized in the following table. The table lists the name of the data type, its notation, and a description of an element of the data type.

Data Type	Notation	Description
Boolean	$\mathbb{B}$	An element of {true, false}.
Character	char	A single symbol or digit.
Double	$\mathbb{R}$	Any number in (-∞, ∞).
Integer	$\mathbb{Z}$	A number without a fractional component in (-∞, ∞).
Unsigned integer	$\mathbb{Z}^+$	A number without a fractional component in $[0, \infty)$.

Chipmunk2D also uses non-primitive data types such as arrays (not to be confused with the Array object of the Sequence Data Structure Module), enumerations, pointers (references), strings, structures, unions. These are summarized in the following table.

Data Type	Notation	Description
Array	array(T)	A list of a given data type T.
Enumeration	enum	A data type containing named, constant values.
Pointer	T^*	A reference to an object of data type T.
String	string/array(char)	An array of characters.
Structure	struct	A data type that can store multiple fields of different data types in one variable.
Union	union	Similar to a structure, but only one field can contain a value at any given time.

Finally, Chipmunk2D uses two more important type-related concepts: void and function pointers. Void is not a data type in itself; however, functions that do not return any value are assigned a return type of void, and void pointers, denoted by void*, are used for references to objects of an unspecified data type. Chipmunk2D also allows passing functions to other functions through the use of function pointers, which hold references to function definitions. Each function pointer is denoted by the name of their function type and is defined by a specific function signature, such as:

$$\text{Function Type} : \text{Arg}_1 \times \text{Arg}_2 \times ... \times \text{Arg}_n \rightarrow \text{Return Type}$$

For example, an inequality operator would have the signature "Inequality : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B}$".

3 Module Hierarchy

To view the Module Hierarchy, please refer to the Module Hierarchy section of the MG.

4 MIS of the Rigid Body Module

4.1 Module Name: Body

4.2 Uses

Shape Module, Space Module, Arbiter Module, Control Module, Vector Module, Transform Matrix Module, Spatial Index Module, Sequence Data Structure Module

4.3 Interface Syntax

4.3.1 Exported Data Types

BodyType: enum
Body: struct
PositionFunc : Body* $\times \mathbb{R} \rightarrow$ void
VelocityFunc : Body* $\times$ Vector $\times \mathbb{R} \rightarrow$ void
ShapeIteratorFunc : Body* $\times$ Shape* $\times$ void* $\rightarrow$ void
ArbiterIteratorFunc : Body* $\times$ Shape* $\times$ void* $\rightarrow$ void

4.3.2 Exported Access Programs

Name	In	Out	Exceptions
bodyAlloc	-	Body*	-
bodyInit	Body*, double, double	Body*	-

newBody	double, double	Body*	NaNMass ∨ NaNMoment ∨ NegativeMass ∨ NegativeMoment ∨ InfiniteMass
newStaticBody	-	Body*	-
bodyDestroy	Body*	-	-
bodyGetType	Body*	BodyType	-
bodyAccumulateMassFromShapes	Body*	-	IllegalBody
bodyGetSpace	Body*	Space*	-
bodyGetMass	Body*	double	-
bodyGetMoment	Body*	double	-
bodyGetRotation	Body*	Vector	-
bodyGetPosition	Body*	Vector	-
bodyGetCenterOfMass	Body*	Vector	-
bodyGetVelocity	Body*	Vector	-
bodyGetForce	Body*	Vector	-
bodyGetAngle	Body*	double	-
bodyGetAngularVelocity	Body*	double	-
bodyGetTorque	Body*	double	-
bodySetType	Body*, BodyType	-	IllegalBody
bodySetMass	Body*, double	-	StaticBodyMass ∨ NegativeMass ∨ InfiniteMass
bodySetMoment	Body*, double	-	NegativeMoment
bodySetPosition	Body*, Vector	-	IllegalBody
bodySetCenterOfMass	Body*, Vector	-	IllegalBody
bodySetVelocity	Body*, Vector	-	IllegalBody
bodySetForce	Body*, Vector	-	IllegalBody
bodySetAngle	Body*, double	-	IllegalBody
bodySetAngularVelocity	Body*, double	-	IllegalBody
bodySetTorque	Body*, double	-	IllegalBody
bodySetPositionFunc	Body*, PositionFunc	-	-

bodySetVelocityFunc	Body*, VelocityFunc	-	-
bodyAddShape	Body*, Shape*	-	-
bodyRemoveShape	Body*, Shape*	-	-
bodyUpdatePosition	Body*, double	-	IllegalBody
bodyUpdateVelocity	Body*, Vector, double	-	IllegalBody
bodyKineticEnergy	Body*	double	-
bodyEachShape	Body*, ShapeIteratorFunc, void*	-	-
bodyEachArbiter	Body*, ArbiterIteratorFunc, void*	-	-

4.4 Interface Semantics

4.4.1 State Variables

BodyType $\in$ {DYNAMIC_BODY, STATIC_BODY}

Body:

type: BodyType
positionFunc: PositionFunc
velocityFunc: VelocityFunc
mass: $\mathbb{R}$
massInv: $\mathbb{R}$
moment: $\mathbb{R}$
momentInv: $\mathbb{R}$
com: Vector
pos: Vector
vel: Vector
force: Vector
angle: $\mathbb{R}$
avel: $\mathbb{R}$
torque: $\mathbb{R}$
velBias: Vector
avelBias: $\mathbb{R}$
transform: Transform
space: Space*
shapeList: Shape*
arbiterList: Arbiter*

4.4.2 State Invariant

For dynamic bodies, the following invariants apply. Any value in $\mathbb{R}$ means that it must be a valid and finite real number:

- Body.mass $\in [0, \infty)$
- Body.moment $\in [0, \infty)$
- $|\text{Body.pos.x}| \in \mathbb{R} \wedge |\text{Body.pos.y}| \in \mathbb{R}$
- $|\text{Body.vel.x}| \in \mathbb{R} \wedge |\text{Body.vel.y}| \in \mathbb{R}$

- $|\text{Body.force.x}| \in \mathbb{R} \wedge |\text{Body.force.y}| \in \mathbb{R}$
- $|\text{Body.angle}| \in \mathbb{R}$
- $|\text{Body.avel}| \in \mathbb{R}$
- $|\text{Body.torque}| \in \mathbb{R}$

4.4.3 Assumptions

bodyAlloc, or newBody, or newStaticBody are called before any other access program. All input pointers are also assumed to be non-null.

4.4.4 Access Program Semantics

bodyAlloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	None.
	Output:	bodyAlloc heap-allocates a new Body object and returns a pointer to it as output.
bodyInit:	**Input:**	bodyInit accepts a Body pointer and two double values as inputs.
	Exceptions:	None.
	Transition:	bodyInit allocates a new Body and initializes its mass and moment with the input values. Other fields are zero-initialized and kinematic functions are set to default ones.
	Output:	bodyInit returns a pointer to the initialized Body.
newBody:	**Input:**	newBody accepts two double values as inputs.
	Exceptions:	newBody may throw a NaNMass, NaNMoment, NegativeMass or NegativeMoment exception when the user provides NaN or negative values for input, or an InfiniteMass exception when the user provides an infinite value for the first input double.
	Transition:	newBody will allocate a new Body, initialize it with the input and default values, and set it to a dynamic body.
	Output:	newBody returns a pointer to the new Body.

newStaticBody:	**Input:**	None.
	Exceptions:	None.
	Transition:	newStaticBody creates a new Body and sets it to a static body.
	Output:	newStaticBody returns a pointer to the new Body.
bodyDestroy:	**Input:**	bodyDestroy accepts a Body pointer.
	Exceptions:	None.
	Transition:	bodyDestroy frees the input Body object.
	Output:	None.
bodyGet:	**Input:**	Each bodyGet function accepts a Body pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each bodyGet function returns the value of their corresponding parameter.
bodySet:	**Input:**	Each bodySet function accepts a Body pointer and their corresponding value as inputs.
	Exceptions:	Various, see Section 4.3.2. The IllegalBody exception occurs when any invariant in 4.4.2 is violated.
	Transition:	Each bodySet function will modify the state of their corresponding parameter to the input value, if valid. bodySetMass and bodySetMoment will also modify the inverse values of the parameters. bodySetType will reset the Body's mass, moment and velocities if changed to a static type, or recalculate its mass from attached Shapes if changed to a dynamic type. It also updates any associated Space accordingly.
	Output:	None.
bodyAccumulate Mass FromShapes	**Input:**	bodyAccumulateMassFromShapes accepts a Body pointer as input.

	Exceptions:	bodyAccumulateMassFromShapes may throw an IllegalBody exception if the Body violates any invariant in 4.4.2 after the transition is complete.
	Transition:	The function recalculates the mass, moment and centre of mass of the Body based on the masses, moments and centres of mass of Shapes associated with it. It will modify the mass and moment inverses accordingly, realign the Body's position in Space, and check that it satisfies all invariants.
	Output:	None.
bodyAddShape:	**Input:**	bodyAddShape accepts a Body pointer and a Shape pointer as inputs.
	Exceptions:	None.
	Transition:	The function will add the input Shape to the input Body's list of Shapes and recalculate the Body's mass accordingly.
	Output:	None.
bodyRemove Shape:	**Input:**	bodyRemoveShape accepts a Body pointer and a Shape pointer as inputs.
	Exceptions:	None.
	Transition:	The function will remove the input Shape from the Body's list of Shapes and, if the Body is dynamic, recalculate its mass accordingly.
	Output:	None.
bodyUpdate Position:	**Input:**	bodyUpdatePosition accepts a Body pointer and a double value as inputs.
	Exceptions:	bodyUpdatePosition may throw an IllegalBody exception if the Body violates any invariant in 4.4.2 after the transition is complete.
	Transition:	bodyUpdatePosition will update the Body's position and angle based on its linear and angular velocities, respectively, their bias values, and the timestep, which is the second input value. It then resets the bias values and checks if any invariant has been violated.
	Output:	None.

bodyUpdate Velocity:	**Input:**	bodyUpdateVelocity accepts a Body pointer, a Vector and a double value as its input.
	Exceptions:	bodyUpdateVelocity may throw an IllegalBody exception if the Body violates any invariant in 4.4.2 after the transition is complete.
	Transition:	bodyUpdateVelocity will update the Body's velocities using the input gravity Vector, forces and torques applied, and the input double value for the timestep. At the end, it resets the Body's force and torque and checks if any invariants have been violated.
	Output:	None.
bodyKinetic Energy:	**Input:**	bodyKineticEnergy accepts a Body pointer as input.
	Exceptions:	None.
	Transition:	bodyKineticEnergy will calculate the Body's kinetic energy based on its mass, moment, and linear and angular velocities.
	Output:	bodyKineticEnergy returns a double value representing the kinetic energy.
bodyEach:	**Input:**	Each bodyEach function accepts a Body pointer, a function pointer to the corresponding iterator, and a void pointer as inputs.
	Exceptions:	None.
	Transition:	Each bodyEach function will iterate through the Body's Shapes or Arbiters, depending on the function's corresponding parameter, and apply the input function to each object in the list, using the data (void pointer) from the third input value.
	Output:	None.

4.4.5 Local Functions

vectAssert NaN:	**Input:**	vectAssertNaN accepts a Vector and a string as inputs.

	Exceptions:	vectAssertNaN throws an exception if the input Vector has NaN values for its fields.
	Transition:	vectAssertNaN checks if the fields of the input Vector are valid numbers. If this test fails, it prints the input string as an error message. Called by vectAssertSane.
	Output:	None.
vectAssert Infinite:	**Input:**	vectAssertInfinite accepts a Vector and a string as inputs.
	Exceptions:	vectAssertInfinite throws an exception if the input Vector has infinite values for its fields.
	Transition:	vectAssertInfinite checks if the fields of the input Vector are finite. If this test fails, it prints the input string as an error message. Called by vectAssertSane.
	Output:	None.
vectAssert Sane:	**Input:**	vectAssertSane accepts a Vector and a string as inputs.
	Exceptions:	vectAssertSane throws an exception if the input Vector has NaN or infinite values for its fields.
	Transition:	vectAssertSane checks if the fields of the input Vector are valid and finite double values. If this test fails, it prints the input string as an error message. Called by assertSaneBody.
	Output:	None.
assertSane Body:	**Input:**	assertSaneBody accepts a Body pointer as input.
	Exceptions:	assertSaneBody may throw an IllegalBody exception if the Body violates any invariant in 4.4.2 after the transition is complete.
	Transition:	assertSaneBody checks if the input Body satisfies all state invariants, and prints various error messages depending on the first invariant found to be violated. Called by various functions in Section 4.4.4.
	Output:	None.

bodySet Transform:	**Input:**	bodySetTransform accepts a Body pointer, a Vector, and a double value as inputs.
	Exceptions:	None.
	Transition:	bodySetTransform mutates the input Body's transformation matrix (used to obtain its local position) using the input position Vector and a rotation vector converted from the given angle (last input value). Called by bodySetPosition and bodySetAngle.
	Output:	None.

5 MIS of the Shape Module

5.1 Module Name: Shape

5.2 Uses

Rigid Body Module, Space Module, Arbiter Module, Control Module, Vector Module, Bounding Box Module, Transform Matrix Module

5.3 Interface Syntax

5.3.1 Exported Constants

MAGIC_EPSILON: $\mathbb{R}$
MAGIC_EPSILON := 1×10^{-5}

POLY_SHAPE_INLINE_ALLOC: $\mathbb{Z}^+$
POLY_SHAPE_INLINE_ALLOC := 6

5.3.2 Exported Data Types

ShapeType: enum
ShapeMassInfo: struct
ShapeClass: struct
Shape: struct
ShapeCacheDataImpl : Shape* $\times$ Transform $\rightarrow$ BB
ShapeDestroyImpl : Shape* $\rightarrow$ void

5.3.3 Exported Access Programs

Name	In	Out	Exceptions
shapeInit	Shape*, ShapeClass*, Body*, ShapeMassInfo	Shape*	-
shapeDestroy	Shape*	-	-
shapeGetSpace	Shape*	Space*	-
shapeGetBody	Shape*	Body*	-
shapeGetMass	Shape*	double	-
shapeGetDensity	Shape*	double	-
shapeGetMoment	Shape*	double	-
shapeGetArea	Shape*	double	-
shapeGetCenterOfMass	Shape*	Vector	-
shapeGetBB	Shape*	BB	-
shapeGetElasticity	Shape*	double	-
shapeGetFriction	Shape*	double	-
shapeGetSurfaceVelocity	Shape*	Vector	-
shapeGetCollisionType	Shape*	CollisionType	-
shapeSetMass	Shape*, double	-	IllegalBody
shapeSetDensity	Shape*, double	-	IllegalBody
shapeSetElasticity	Shape*, double	-	NegativeElasticity
shapeSetFriction	Shape*, double	-	NegativeFriction
shapeSetSurfaceVelocity	Shape*, Vector	-	-
shapeSetCollisionType	Shape*, CollisionType	-	-
shapeCacheBB	Shape*	BB	-
shapeUpdate	Shape*, Transform	BB	-

5.4 Interface Semantics

5.4.1 State Variables

ShapeType $\in$ {CIRCLE_SHAPE, SEGMENT_SHAPE, POLY_SHAPE, NUM_SHAPES}

ShapeMassInfo:

mass: $\mathbb{R}$
moment: $\mathbb{R}$
com: Vector
area: $\mathbb{R}$

ShapeClass:

type: ShapeType
cacheData: ShapeCacheDataImpl
destroy: ShapeDestroyImpl

Shape:

klass: ShapeClass*
space: Space*
body: Body*
massInfo: ShapeMassInfo
bb: BB
elast: $\mathbb{R}$
fric: $\mathbb{R}$
surfaceVel: Vector
type: CollisionType
next: Shape*
prev: Shape*
hashId: HashValue

5.4.2 State Invariants

Shape.elast ≥ 0
Shape.fric ≥ 0

5.4.3 Assumptions

All input pointers are assumed to be non-null. Also see 5.8.2, 5.12.2 and 5.16.2.

5.4.4 Access Program Semantics

shapeInit:	**Input:**	shapeInit accepts a Shape pointer, ShapeClass pointer, Body pointer and a ShapeMassInfo structure as inputs.
	Exceptions:	None.
	Transition:	shapeInit initializes the input Shape. It sets the Shape's class, body and mass information to the input parameters, and zero-initializes all other variables.
	Output:	shapeInit returns a pointer to the initialized Shape as output.
shapeDestroy:	**Input:**	shapeDestroy accepts a Shape pointer as input.
	Exceptions:	None.

	Transition:	shapeDestroy frees the input Shape object.
	Output:	None.
shapeGet:	**Input:**	Each shapeGet function accepts a Shape pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each shapeGet function returns the value of their corresponding parameter.
shapeSet:	**Input:**	Each shapeSet function accepts a Shape pointer and their corresponding parameter as inputs.
	Exceptions:	Various, see Section 5.3.3.
	Transition:	Each shapeSet function will set the corresponding parameter to the input value. shapeSetMass sets the mass of the Shape's mass information and recalculates the mass of its associated Body accordingly.
	Output:	None.
shapeCacheBB:	**Input:**	shapeCacheBB accepts a Shape pointer as input.
	Exceptions:	None.
	Transition:	shapeCacheBB updates the input Shape using the Transform matrix of its associated Body and modifies its bounding box (BB).
	Output:	shapeCacheBB returns the new BB as output.
shapeUpdate:	**Input:**	shapeUpdate accepts a Shape pointer and a Transform matrix as inputs.
	Exceptions:	None.
	Transition:	shapeUpdate will call the cacheData function in the input Shape's class using the given parameters and modify the Shape's BB.
	Output:	shapeUpdate returns the BB returned by the cacheData function as output.

5.5 Submodule Name: CircleShape

5.6 Uses

Rigid Body Module, Shape Module, Control Module, Vector Module, Bounding Box Module, Transform Matrix Module

5.7 Interface Syntax

5.7.1 Exported Data Types

CircleShape: struct

5.7.2 Exported Access Programs

Name	In	Out	Exceptions
circleShapeAlloc	-	CircleShape*	-
circleShapeInit	CircleShape*, Body*, double, Vector	CircleShape*	-
circleShapeNew	Body*, double, Vector	Shape*	-
circleShapeGetRadius	Shape*	double	NotCircleShape
circleShapeGetOffset	Shape*	Vector	NotCircleShape
circleShapeSetRadius	Shape*, double	-	NotCircleShape $\vee$ IllegalBody
circleShapeSetOffset	Shape*, Vector	-	NotCircleShape $\vee$ IllegalBody
momentForCircle	double, double, double, Vector	double	-
areaForCircle	double, double	double	-

5.8 Interface Semantics

5.8.1 State Variables

CircleShape:

shape: Shape
center: Vector
tcenter: Vector
radius: $\mathbb{R}$

Note that *center* is the centroid of the circle, and *tcenter* is the transformed centroid in global coordinates.

5.8.2 Assumptions

circleShapeAlloc or circleShapeNew have been called before any other access programs. All input pointers are also assumed to be non-null.

5.8.3 Access Program Semantics

circleShapeAlloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	None.
	Output:	circleShapeAlloc heap-allocates a new CircleShape object and returns a pointer to it as output.
circleShapeInit:	**Input:**	circleShapeInit accepts a CircleShape pointer, a Body pointer, a double and a Vector as inputs.
	Exceptions:	None.
	Transition:	circleShapeInit initializes the input CircleShape. It sets the radius to the input double, the center to the input Vector, and then initializes the rest of the variables using shapeInit and the input Body.
	Output:	circleShapeInit returns a pointer to the initialized CircleShape.
circleShapeNew:	**Input:**	circleShapeNew accepts a Body pointer, a double and a Vector as inputs.
	Exceptions:	None.
	Transition:	circleShapeNew allocates and initializes a new CircleShape object using the input paramters.
	Output:	circleShapeNew returns a pointer to the new CircleShape.
circleShapeGet:	**Input:**	Each circleShapeGet function accepts a Shape pointer as input.
	Exceptions:	Each circleShapeGet function may throw a NotCircleShape exception if the input Shape pointer is not of the CircleShape class.

	Transition:	None.
	Output:	Each circleShapeGet function returns the value of their corresponding parameter.
circleShapeSet:	**Input:**	Each circleShapeSet function accepts a Shape pointer and their corresponding parameter as inputs.
	Exceptions:	Each circleShapeSet function may throw a NotCircleShape exception if the input Shape pointer is not of the CircleShape class, or if the Body associated with the Shape violates an invariant in 4.4.2 after the transitions are complete.
	Transition:	Each circleShapeSet function sets their corresponding parameter with the input value, updates the mass information of the Shape and recalculates the mass of its associated Body.
	Output:	None.
momentForCircle:	**Input:**	momentForCircle accepts three doubles for mass, inner radius and outer radius, and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	momentForCircle returns the calculated moment from the input parameters as a double.
areaForCircle:	**Input:**	areaForCircle accepts two double values for the inner radius and outer radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	areaForCircle returns the calculated area from the input parameters as a double.

5.8.4 Local Constants

CircleShapeClass: ShapeClass
CircleShapeClass := {CIRCLE_SHAPE, circleShapeCacheData, NULL}

5.8.5 Local Functions

circleShapeCacheData:

Input: circleShapeCacheData accepts a CircleShape pointer and a Transform matrix as inputs.

Exceptions: None.

Transition: circleShapeCacheData updates the transformed center of the input CircleShape using the input Transform matrix and generates a new BB with the CircleShape's properties. Default cacheData method of the CircleShapeClass.

Output: circleShapeCacheData returns the new BB as output.

circleShapeMassInfo:

Input: circleShapeMassInfo accepts two double values for mass and radius and a Vector as inputs.

Exceptions: None.

Transition: None.

Output: circleShapeMassInfo is a convenience constructor which returns a new ShapeMassInfo structure for CircleShapes, initialized using the input values.

5.9 Submodule Name: SegmentShape

5.10 Uses

Rigid Body Module, Shape Module, Polygon Module, Vector Module, Bounding Box Module, Transform Matrix Module

5.11 Interface Syntax

5.11.1 Exported Data Types

SegmentShape: struct

5.11.2 Exported Access Programs

Name	In	Out	Exceptions
segmentShapeAlloc	-	SegmentShape*	-
segmentShapeInit	SegmentShape*, Body*, Vector, Vector, double	SegmentShape*	-

segmentShapeNew	Body*, Vector, Vector, double	Shape*	-
segmentShapeGetA	Shape*	Vector	NotSegmentShape
segmentShapeGetB	Shape*	Vector	NotSegmentShape
segmentShapeGetNormal	Shape*	Vector	NotSegmentShape
segmentShapeGetRadius	Shape*	double	NotSegmentShape
segmentShapeSetNeighbors	Shape*, Vector, Vector	-	NotSegmentShape
segmentShapeSetEndpoints	Shape*, Vector, Vector	-	NotSegmentShape $\vee$ IllegalBody
segmentShapeSetRadius	Shape*, double	-	NotSegmentShape $\vee$ IllegalBody
momentForSegment	double, Vector, Vector, double	double	-
areaForSegment	Vector, Vector, double	double	-

5.12 Interface Semantics

5.12.1 State Variables

SegmentShape:

shape: Shape
a: Vector
b: Vector
n: $\mathbb{R}$
ta: Vector
tb: Vector
tn: Vector
radius: $\mathbb{R}$
aTangent: Vector
bTangent: Vector

Note that a and b are the endpoints of the segment, and n is the normal. ta, tb and tn are the transformed endpoints in global coordinates.

5.12.2 Assumptions

segmentShapeAlloc or segmentShapeNew, have been called before any other access programs. All input pointers are also assumed to be non-null.

5.12.3 Access Program Semantics

segmentShape Alloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	None.
	Output:	segmentShapeAlloc heap-allocates a new SegmentShape object and returns a pointer to it as output.
segmentShape Init:	**Input:**	segmentShapeInit accepts a SegmentShape pointer, a Body pointer, two Vectors and a double as inputs.
	Exceptions:	None.
	Transition:	segmentShapeInit initializes the input SegmentShape. It sets the endpoints to the given Vectors, the radius to the given double, and initializes the rest with shapeInit and the input Body.
	Output:	segmentShapeInit returns a pointer to the initialized SegmentShape.
segmentShape New:	**Input:**	segmentShapeNew accepts a Body pointer, two Vectors and a double as inputs.
	Exceptions:	None.
	Transition:	segmentShapeNew allocates and initializes a new SegmentShape object using the input parameters.
	Output:	segmentShapeNew returns a pointer to the new SegmentShape.
segmentShape Get:	**Input:**	Each segmentShapeGet function accepts a Shape pointer as input.
	Exceptions:	Each segmentShapeGet function may throw a NotSegmentShape exception if the input Shape pointer is not of the SegmentShape class.
	Transition:	None.
	Output:	Each segmentShapeGet function returns the value of their corresponding parameter.

segmentShape Set:	**Input:**	Each segmentShapeSet function accepts a Shape pointer and their corresponding parameter as inputs. In particular, segmentShapeSetNeighbors and segmentShapeSetEndpoints accept two Vectors as inputs.
	Exceptions:	Each segmentShapeSet function may throw a NotSegmentShape exception if the input Shape pointer is not of the SegmentShape class. segmentShapeSetEndpoints and segmentShapeSetRadius may throw an IllegalBody exception if the Body associated with the Shape violates an invariant in 4.4.2 after the transitions are complete.
	Transition:	Each segmentShapeSet function sets their corresponding parameter with the input value. In addition, segmentShapeSetEndpoints and segmentShapeSetRadius update the mass information of the Shape and recalculate the mass of the associated Body.
	Output:	None.
momentFor Segment:	**Input:**	momentForSegment accepts a double for mass, two Vectors for endpoints, and another double for radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	momentForSegment returns the calculated moment from the input parameters as a double.
areaForSegment:	**Input:**	areaForSegment accepts two Vectors for endpoints and a double for radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	areaForSegment returns the calculated area from the input parameters as a double.

5.12.4 Local Constants

SegmentShapeClass: ShapeClass
SegmentShapeClass := {SEGMENT_SHAPE, segmentShapeCacheData, NULL}

5.12.5 Local Functions

segmentShape CacheData:	**Input:**	segmentShapeCacheData accepts a SegmentShape pointer and a Transform matrix as inputs.
	Exceptions:	None.
	Transition:	segmentShapeCacheData updates the transformed endpoints and normal for the input SegmentShape using the input Transform matrix and generates a new BB with the SegmentShape's properties. Default cacheData method for the SegmentShapeClass.
	Output:	segmentShapeCacheData returns the new BB as output.
segmentShape MassInfo:	**Input:**	segmentShapeMassInfo accepts a double for mass, two Vectors for endpoints and a double for radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	segmentShapeMassInfo is a convenience constructor that returns a new ShapeMassInfo structure for SegmentShapes, initialized using the input values.

5.13 Submodule Name: PolyShape

5.14 Uses

Rigid Body Module, Shape Module, Segment Module, Control Module, Vector Module, Bounding Box Module, Transform Matrix Module

5.15 Interface Syntax

5.15.1 Exported Data Types

SplittingPlane: struct
PolyShape: struct

5.15.2 Exported Access Programs

Name	In	Out	Exceptions
polyShapeAlloc	-	PolyShape*	-

Raw	PolyShape*, Body*, int, Vector*, double	PolyShape*	-
polyShapeInit	PolyShape*, Body*, int, Vector*, double, Transform	PolyShape*	-
boxShapeInit	PolyShape*, Body*, double, double, double	PolyShape*	-
boxShapeInit2	PolyShape*, Body*, double, BB	PolyShape*	-
polyShapeNew	Body*, int, Vector*, double	Shape*	-
polyShapeNewRaw	Body*, int, Vector*, double, Transform	Shape*	-
boxShapeNew	Body*, double, double, double	Shape*	-
boxShapeNew2	Body*, double, BB	Shape*	-
polyShapeGetCount	Shape*	int	NotPolyShape
polyShapeGetVert	Shape*, int	Vector	NotPolyShape $\vee$ IndexOutOf-Bounds
polyShapeGetRadius	Shape*	double	NotPolyShape
polyShapeSetVerts	Shape*, int, Vector*, Transform	-	NotPolyShape $\vee$ IllegalBody
polyShapeSetVertsRaw	Shape*, int, Vector*	-	NotPolyShape $\vee$ IllegalBody
polyShapeSetRadius	Shape*, double	-	NotPolyShape
momentForPoly	double, int, Vector*, Vector, double	double	-
areaForPoly	int, Vector*, double	double	-
centroidForPoly	int, Vector*	Vector	-

5.16 Interface Semantics

5.16.1 State Variables

SplittingPlane:

v0: Vector
n: Vector

PolyShape:

shape: Shape
radius: $\mathbb{R}$
count: $\mathbb{Z}$
planes: SplittingPlane*
_planes: array(SplittingPlane)

5.16.2 Assumptions

polyShapeAlloc, or polyShapeNew/polyShapeNewRaw, or boxShapeNew/boxShapeNew2, have been called before any other access programs. All input pointers are also assumed to be non-null.

5.16.3 Access Program Semantics

polyShapeAlloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	None.
	Output:	polyShapeAlloc heap-allocates a new PolyShape object and returns a pointer to it as output.
polyShapeInit:	**Input:**	polyShapeInit accepts a PolyShape pointer, a Body pointer, an integer, a pointer to a Vector array, a double and a Transform matrix as inputs.
	Exceptions:	None.
	Transition:	polyShapeInit transforms each vertex from the input array with the input Transform matrix, places the resultant vertices in a new array, calculates the size of the convex hull containing the new vertices and initializes the input PolyShape using this array, the hull size and the remaining parameters.
	Output:	polyShapeInit returns a pointer to the initialized PolyShape.

polyShapeInit Raw:	**Input:**	polyShapeInitRaw accepts a PolyShape pointer, a Body pointer, an integer, a pointer to a Vector array and a double as inputs.
	Exceptions:	None.
	Transition:	polyShapeInitRaw initializes the input PolyShape using shapeInit and the input parameters, sets its vertices to the given array and integer (which represents the length of the array), and sets its radius to the input double.
	Output:	polyShapeInitRaw returns a pointer to the initialized PolyShape.
boxShapeInit:	**Input:**	boxShapeInit accepts a PolyShape pointer, Body pointer and three doubles as inputs.
	Exceptions:	None.
	Transition:	boxShapeInit calculates values for half-width and half-height using the last two input doubles as width and height, respectively. It then initializes the input PolyShape using a new BB generated from the calculated half-dimensions and the remaining parameters.
	Output:	boxShapeInit returns a pointer to the initialized PolyShape.
boxShapeInit2:	**Input:**	boxShapeInit2 accepts a PolyShape pointer, Body pointer, a double and a BB as inputs.
	Exceptions:	None.
	Transition:	boxShapeInit2 creates a Vector array containing the vertices of the box, determined from the input BB. It then initializes the input PolyShape as a box using the array and number of vertices, as well as the remaining parameters.
	Output:	boxShapeInit2 returns a pointer to the initialized PolyShape.
polyShapeNew:	**Input:**	Each polyShapeNew function accepts a Body pointer, an integer, a pointer to a Vector array and a double as inputs. In addition, polyShapeNew (not Raw) accepts a Transform matrix as its last input.
	Exceptions:	None.

	Transition:	Each polyShapeNew function allocates and initializes a new PolyShape object using the input parameters.
	Output:	Each polyShapeNew function returns a pointer to the new PolyShape.
boxShapeNew:	**Input:**	Each boxShapeNew function accepts a Body pointer and a double as inputs. In addition, boxShapeNew accepts two additional doubles, while boxShapeNew2 accepts an additional BB as input.
	Exceptions:	None.
	Transition:	Each boxShapeNew function allocates and initializes a new PolyShape object as a box using the input parameters.
	Output:	Each boxShapeNew function returns a pointer to the new PolyShape.
polyShapeGet:	**Input:**	Each polyShapeGet function accepts a Shape pointer as input. polyShapeGetVert also accepts an additional integer as input.
	Exceptions:	Each polyShapeGet function may throw a NotPolyShape exception if the input Shape pointer is not of the PolyShape class. polyShapeGetVert may also throw an exception if the input integer is greater than or equal to the number of vertices of the input Shape.
	Transition:	None.
	Output:	Each polyShapeGet function returns the value of their corresponding parameter.
polyShapeSet:	**Input:**	Each polyShapeSet function accepts a Shape pointer and their corresponding parameter as inputs. Specifically, each polyShapeSetVerts function accepts an integer (for the number of vertices) and a pointer to a Vector array (holding the vertices) as inputs, and polyShapeSetVerts (not Raw) accepts an additional Transform matrix.

Exceptions: Each polyShapeSet function may throw a NotPolyShape exception if the input Shape pointer is not of the PolyShape class. Each polyShapeSetVerts function may throw an IllegalBody exception if the Body associated with the Shape violates an invariant in 4.4.2 after the transitions are complete.

Transition: Each polyShapeSet function sets their corresponding parameter with the input value. More specifically, polyShapeVerts transforms the vertices in the input array with the input Transform matrix, places the resultant vertices in a new array, determines the size of the convex hull containing these vertices, and calls polyShapeSetVertsRaw with the new array and hull size. polyShapeVertsRaw frees the current vertices of the input PolyShape, sets its new vertices, updates the mass information of the Shape and recalculates the mass of the associated Body.

Output: None.

momentForPoly:

Input: momentForPoly accepts a double for mass, an integer for number of vertices, a pointer to a Vector array containing these vertices, a Vector for offset, and a double for radius as inputs.

Exceptions: None.

Transition: None.

Output: momentForPoly returns the calculated moment from the input parameters as a double.

areaForPoly:

Input: areaForPoly accepts an integer for number of vertices, a pointer to a Vector array containing these vertices, and a double for radius as inputs.

Exceptions: None.

Transition: None.

Output: areaForPoly returns the calculated area from the input parameters as a double.

centroidForPoly:

Input: centroidForPoly accepts an integer for number of vertices and a pointer to a Vector array containing these vertices as inputs.

	Exceptions:	None.
	Transition:	None.
	Output:	centroidForPoly returns the calculated centroid from the input parameters as a Vector.

5.16.4 Local Constants

PolyShapeClass: ShapeClass
PolyShapeClass := {POLY_SHAPE, polyShapeCacheData, polyShapeDestroy}

5.16.5 Local Functions

polyShape Destroy:	**Input:**	polyShapeDestroy accepts a PolyShape pointer as input.
	Exceptions:	None.
	Transition:	polyShapeDestroy frees the input PolyShape and its associated array of vertices. Default destroy method for the PolyShapeClass.
	Output:	None.
polyShape CacheData:	**Input:**	polyShapeCacheData accepts a PolyShape pointer and a Transform matrix as inputs.
	Exceptions:	None.
	Transition:	polyShapeCacheData transforms each vertex of the input PolyShape using the input matrix, calculates the extreme points of the shape, and updates the PolyShape's BB to a new BB generated from the calculated extremes. Default cacheData method for the PolyShapeClass.
	Output:	polyShapeCacheData returns the new BB as output.
setVerts:	**Input:**	setVerts accepts a PolyShape pointer, an integer, and a pointer to a Vector array as inputs.
	Exceptions:	None.

	Transition:	setVerts sets the input PolyShape's number of vertices to the input integer. If this is less than or equal to POLY_SHAPE_INLINE_ALLOC, the PolyShape uses its default _planes array for its vertices. Otherwise, it heap-allocates a new array with the length of the input integer. Finally, the function iterates through the planes array and sets the vertices and their calculated edge normals from the input array. Called by polyShapeInitRaw and polyShapeSetVertsRaw to mutate vertices.
	Output:	None.
polyShape MassInfo:	**Input:**	polyShapeMassInfo accepts a double for mass, an integer for number of vertices, a pointer to a Vector array containing these vertices, and a double for radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	polyShapeMassInfo is a convenience constructor that returns a new ShapeMassInfo structure for PolyShapes, initialized using the input values.

6 MIS of the Space Module

6.1 Module Name: Space

6.2 Uses

Rigid Body Module, Shape Module, Arbiter Module, Control Module, Vector Module, Spatial Index Module, Sequence Data Structure Module, Linked Data Structure Module, Associative Data Structure Module

6.3 Interface Syntax

6.3.1 Exported Constants

collisionHandlerDoNothing: CollisionHandler
collisionHandlerDoNothing := {WILDCARD_COLLISION_TYPE, WILDCARD_COLLISION_TYPE, alwaysCollide, alwaysCollide, doNothing, doNothing, NULL}

CONTACTS_BUFFER_SIZE: $\mathbb{Z}^+$
CONTACTS_BUFFER_SIZE := (BUFFER_BYTES - sizeof(ContactBufferHeader)) / sizeof(Contact)

6.3.2 Exported Data Types

Space: struct
PostStepCallback: struct
CollisionHandler: struct
ArbiterFilterContext: struct
SpaceShapeContext: struct
ContactBufferHeader: struct
ContactBuffer: struct
SpaceArbiterApplyImpulseFunc : Arbiter* $\rightarrow$ void
CollisionBeginFunc : Arbiter* $\times$ Space* $\times$ void* $\rightarrow \mathbb{B}$
CollisionPreSolveFunc : Arbiter* $\times$ Space* $\times$ void* $\rightarrow \mathbb{B}$
CollisionPostSolveFunc : Arbiter* $\times$ Space* $\times$ void* $\rightarrow$ void
CollisionSeparateFunc : Arbiter* $\times$ Space* $\times$ void* $\rightarrow$ void
PostStepFunc : Space* $\times$ void* $\times$ void* $\rightarrow$ void
SpaceBodyIteratorFunc : Body* $\times$ void* $\rightarrow$ void
SpaceShapeIteratorFunc : Shape* $\times$ void* $\rightarrow$ void

6.3.3 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
spaceAlloc	-	Space*	-
spaceInit	Space*	Space*	-
spaceNew	-	Space*	-
spaceDestroy	Space*	-	-
spaceFree	Space*	-	-
spaceGetIterations	Space*	int	-
spaceGetGravity	Space*	Vector	-
spaceGetCollisionSlop	Space*	double	-
spaceGetCollisionBias	Space*	double	-
spaceGetCollisionPersistence	Space*	Timestamp	-
spaceGetCurrentTimeStep	Space*	double	-
spaceGetStaticBody	Space*	Body*	-
spaceSetIterations	Space*, int	-	InvalidIter
spaceSetGravity	Space*, Vector	-	-
spaceSetCollisionSlop	Space*, double	-	-
spaceSetCollisionBias	Space*, double	-	-

spaceSetCollisionPersistence	Space*, Timestamp	-	-
spaceSetStaticBody	Space*, Body*	-	AttachedStaticBody
spaceIsLocked	Space*	Boolean	-
spaceAddDefaultCollisionHandler	Space*	CollisionHandler*	-
spaceAddCollisionHandler	Space*, CollisionType, CollisionType	CollisionHandler*	-
spaceAddWildcardHandler	Space*, CollisionType	CollisionHandler*	-
spaceAddShape	Space*, Shape*	Shape*	DuplicateShape ∨ AttachedShape ∨ SpaceLocked
spaceAddBody	Space*, Body*	Body*	DuplicateBody ∨ AttachedBody ∨ SpaceLocked
spaceFilterArbiters	Space*, Body*, Shape*	-	-
spaceRemoveShape	Space*, Shape*	-	ShapeNotFound ∨ SpaceLocked
spaceRemoveBody	Space*, Body*	-	MainStaticBody ∨ BodyNotFound ∨ SpaceLocked
spaceContainsShape	Space*, Shape*	Boolean	-
spaceContainsBody	Space*, Body*	Boolean	-
spaceEachBody	Space*, Space-BodyIteratorFunc, void*	-	-
spaceEachShape	Space*, Space-ShapeIteratorFunc, void*	-	-
spaceReindexStatic	Space*	-	SpaceLocked
spaceReindexShape	Space*, Shape*	-	SpaceLocked
spaceReindexShapesForBody	Space*, Body*	-	SpaceLocked
spacePushFreshContactBuffer	Space*	-	-
contactBufferGetArray	Space*	Contact	-

spaceCollideShapes	Shape*, Shape*, CollisionID, Space*	CollisionID	-
spaceArbiterSetFilter	Arbiter*, Space*	Boolean	-
spaceLock	Space*	-	-
spaceUnlock	Space*, Boolean	-	SpaceLockUnderflow
spaceArrayForBodyType	Space*, BodyType	Array*	-
shapeUpdateFunc	Shape*, void*	-	-
spaceStep	Space*, double	-	-

6.4 Interface Semantics

6.4.1 State Variables

Space:

iterations: $\mathbb{Z}$
gravity: Vector
collisionSlop: $\mathbb{R}$
collisionBias: $\mathbb{R}$
collisionPersistence: Timestamp
stamp: Timestamp
curr_dt: $\mathbb{R}$
dynamicBodies: Array*
staticBodies: Array*
shapeIDCounter: HashValue
staticShapes: SpatialIndex*
dynamicShapes: SpatialIndex*
arbiters: Array*
contactBuffersHead: ContactBufferHeader*
cachedArbiters: HashSet*
pooledArbiters: Array*
allocatedBuffers: Array*
locked: $\mathbb{Z}^+$
usesWildcards: $\mathbb{B}$
collisionHandlers: HashSet*
defaultHandler: CollisionHandler
skipPostStep: $\mathbb{B}$
postStepCallbacks: Array*
staticBody: Body*
_staticBody: Body

PostStepCallback:

func: PostStepFunc
key: void*
data: void*

CollisionHandler:

typeA: CollisionType
typeB: CollisionType
beginFunc: CollisionBeginFunc
preSolveFunc: CollisionPreSolveFunc
postSolveFunc: CollisionPostSolveFunc
separateFunc: CollisionSeparateFunc
userData: DataPointer

ArbiterFilterContext:

space: Space*
body: Body*
shape: Shape*

SpaceShapeContext:

func: SpaceShapeIteratorFunc
data: void*

ContactBufferHeader:

stamp: Timestamp
next: ContactBufferHeader*
numContacts: $\mathbb{Z}^+$

ContactBuffer:

header: ContactBufferHeader
contacts: array(Contact)

6.4.2 Assumptions

spaceAlloc or spaceNew are called before any other access programs.

6.4.3 Access Program Semantics

spaceAlloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	None.
	Output:	spaceAlloc heap-allocates a new Space object and returns a pointer to it as output.
spaceInit:	**Input:**	spaceInit accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spaceInit initializes the input Space, allocating new data structures accordingly and zero-initializing all other variables.
	Output:	spaceInit returns a pointer to the initialized Space.

spaceNew:	**Input:**	None..
	Exceptions:	None.
	Transition:	spaceNew allocates and initializes a new Space object.
	Output:	spaceNew returns a pointer to the new Space.
spaceDestroy:	**Input:**	spaceDestroy accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spaceDestroy frees all dynamically-allocated data structures in the input Space and their contents, if necessary.
	Output:	None.
spaceFree:	**Input:**	spaceFree accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spaceFree frees the input Space and all of its dynamically-allocated variables.
	Output:	None.
spaceGet:	**Input:**	Each spaceGet function accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each spaceGet function returns the value of their corresponding parameter.
spaceSet:	**Input:**	Each spaceSet function accepts a Space pointer and their corresponding parameter as input.
	Exceptions:	Various, see 6.3.3. spaceSetStaticBody may throw an AttachedStaticBody exception if the user attempts to change the designated static body while the existing body still has shapes attached to it.
	Transition:	Each spaceSet function sets the value of the corresponding field with the input parameter. For spaceSetStaticBody, it also changes the Body's associated Space to the input Space.
	Output:	None.

spaceIsLocked:	**Input:**	spaceIsLocked accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	spaceIsLocked returns true if the value of the locked variable is positive, and false otherwise.
spaceAdd:	**Input:**	Each spaceAdd function accepts a Space pointer as input. spaceAddCollisionHandler accepts two additional CollisionTypes, while spaceAddWildcardHandler only accepts one additional CollisionType. spaceAddShape and spaceAddBody accepts additional Shape and Body pointers as input, respectively.
	Exceptions:	Various, see 6.3.3. spaceAddShape and spaceAddBody may throw a DuplicateShape/DuplicateBody exception if the object being added already exists in the input Space, an AttachedShape/AttachedBody exception if it is already attached to another Space, or a SpaceLocked exception if the input Space is locked.
	Transition:	Functions that add CollisionHandlers will initialize new handlers depending on the function and input CollisionTypes (or use the default), hash the handler, and add it to the input Space's collision handlers. spaceAddShape and spaceAddBody will add the body to the appropriate spatial index and array of the input Space, respectively, and the former will update the input Shape accordingly.
	Output:	Each spaceAdd function returns the object that has been added as output.
spaceFilter Arbiters:	**Input:**	spaceFilterArbiters accepts Space, Body and Shape pointers as input.
	Exceptions:	None.
	Transition:	spaceFilterArbiters will remove Arbiters that are associated with the input Body and/or Shape from the Space's cached Arbiters.
	Output:	None.
spaceRemove:	**Input:**	Each spaceRemove function accepts a Space pointer and a pointer to the corresponding object as input.

	Exceptions:	Each spaceRemove function may throw a Shape/BodyNotFound exception if the object to be removed does not exist within the input Space or a SpaceLocked exception if the Space is locked. Additionally, spaceRemoveBody will throw a MainStaticBody exception if the user attempts to remove the Space's designated static Body.
	Transition:	Each spaceRemove function will detach the object and the input Space from each other. Additionally, spaceRemoveShape will detach the Shape from its Body.
	Output:	None.
spaceContains:	**Input:**	Each spaceContains function accepts a Space pointer and a pointer to the corresponding object as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each spaceContains function returns true if the input Space contains the input object, and false otherwise.
spaceEach:	**Input:**	Each spaceEach function accepts a Space pointer, a function pointer to the corresponding iterator, and a void pointer as inputs.
	Exceptions:	None.
	Transition:	Each spaceEach function will iterate through the Space's Bodies or Shapes, depending on the function's corresponding parameter, and apply the given function to each object using the data in the input void pointer.
	Output:	None.
spaceReindex:	**Input:**	Each spaceReindex function accepts a Space pointer as input. spaceReindexShape and spaceReindexShapesForBody accepts an additional Shape and Body pointer, respectively.
	Exceptions:	Each spaceReindex function may throw a SpaceLocked exception if the input Space is locked.

	Transition:	spaceReindexStatic reindexes all the static Shapes for the Space. spaceReindexShape will only reindex the input Shape in its spatial index, and spaceReindexShapesForBody will reindex all the shapes attached to the input Body in their respective spatial indices.
	Output:	None.
spacePushFreshContactBuffer:	**Input:**	spacePushFreshContactBuffer accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spacePushFreshContactBuffer allocates a new ContactBufferHeader, initializes it and sets it to the input Space's contact buffer head.
	Output:	None.
contactBufferGetArray:	**Input:**	contactBufferGetArray accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	contactBufferGetArray pushes a fresh ContactBufferHeader if the contact buffer is about to overflow.
	Output:	contactBufferGetArray returns a pointer to an array of Contact structures as output.
spaceCollideShapes:	**Input:**	spaceCollideShapes accepts two Shape pointers, a Collision ID and a Space pointer as inputs.
	Exceptions:	None.
	Transition:	spaceCollideShapes tests if the input Shapes can be collided using queryReject. If it fails, it returns the input ID. Otherwise, it performs collision detection and makes a new CollisionInfo structure. If a collision occurs, the function modifies the number of Contacts for the input Space, updates the Arbiter for the input Shapes, calls the Arbiter's collision handler functions and updates the Arbiter's timestamp. Otherwise, no further transitions are made. In either case, the function returns the ID of the generated CollisionInfo structure.

	Output:	spaceCollideShapes returns a CollisionID as output.
spaceArbiter SetFilter:	**Input:**	spaceArbiterSetFilter accepts one Arbiter and one Space pointer as inputs.
	Exceptions:	None.
	Transition:	spaceArbiterSetFilter caches the input Arbiter if it was uncached and used recently. If the time since the Arbiter was last used exceeds the Space's collision persistence, the function will also remove the Arbiter from the Space and recycle it.
	Output:	spaceArbiterSet returns true if both Bodies attached to the Arbiter are static bodies or if the Arbiter was cached. It returns false if the Arbiter is removed.
spaceLock:	**Input:**	spaceLock accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spaceLock locks the input Space by modifying its locked variable.
	Output:	None.
spaceUnlock:	**Input:**	spaceUnlock accepts a Space pointer and a Boolean value as input.
	Exceptions:	spaceUnlock may throw a SpaceLockUnderflow exception if the locked field of the input Space falls to a negative value during the function's transition.
	Transition:	spaceUnlock will unlock the input Space by modifying its locked variable. If the input Boolean is true, and the Space is set to not skip post-step callbacks, the function will also run those callbacks.
	Output:	None.
spaceArrayFor BodyType:	**Input:**	spaceArrayForBodyType accepts a Space pointer and a BodyType value as inputs.
	Exceptions:	None.
	Transition:	None.

	Output:	spaceArrayForBodyType returns an Array pointer to the input Space's array of bodies, corresponding to the input BodyType.
shapeUpdate Func:	**Input:**	shapeUpdateFunc accepts a Shape and a void pointer as inputs.
	Exceptions:	None.
	Transition:	shapeUpdateFunc calls updates and caches the input Shape's BB.
	Output:	None.
spaceStep:	**Input:**	spaceStep accepts a Space pointer and a double as inputs.
	Exceptions:	None.
	Transition:	spaceStep updates the input Space following the specified timestep (input double). If the timestep is zero, the function exits immediately. Otherwise, it updates the Space's timestamp and current timestep, resets Arbiter lists and locks the Space. While the Space is locked, the function calculates new positions of Bodies in the Space and collides Shapes as necessary, before unlocking the Space without running post-step callbacks. Next, it locks the Space once again, clears cached Arbiters, pre-processes the Arbiters, updates the velocities of Bodies in the Space, applies cached impulses, runs the impulse solver, and then runs post-solve callbacks on the Arbiters. Finally, it unlocks the Space and runs post-step callbacks.
	Output:	None.

6.4.4 Local Constants

collisionHandlerDefault: CollisionHandler
collisionHandlerDefault := {WILDCARD_COLLISION_TYPE, WILDCARD_COLLISION_TYPE, defaultBegin, defaultPreSolve, defaultPostSolve, defaultSeparate, NULL}

6.4.5 Local Functions

default:	**Input:**	Each default function accepts an Arbiter pointer, a Space pointer and a void pointer as inputs.

	Exceptions:	None.
	Transition:	Each default function calls their respective wildcard functions for Shape A and B of the Arbiter. For example, defaultBegin calls arbiterCallWildcardBeginA and arbiterCallWildcardBeginB with the input Arbiter and Space, defaultPreSolve calls arbiterCallWildcardPreSolveA and arbiterCallWildcardPreSolveB, and so on. Part of the default collision handler, collisionHandlerDefault.
	Output:	defaultBegin and defaultPreSolve applies Boolean AND on the result of both wildcard calls and returns the result. defaultPostSolve and defaultSeparate do not return any values.
alwaysCollide:	**Input:**	alwaysCollide takes an Arbiter pointer, a Space pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	alwaysCollide always returns true. Part of collisionHandlerDoNothing that does nothing.
doNothing:	**Input:**	doNothing takes an Arbiter pointer, a Space pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	None. Part of collisionHandlerDoNothing that does nothing.
	Output:	None.

spaceUse WildcardDe-faultHandler:	**Input:**	spaceUseWildcardDefaultHandler accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	The function sets the Space to use wildcards and copies collisionHandlerDefault to the Space's default handler. Called by spaceAddDefaultCollisionHandler.
	Output:	None.
spaceAlloc Contact-Buffer:	**Input:**	spaceAllocContactBuffer accepts a Space pointer as input.
	Exceptions:	None.
	Transition:	spaceAllocContactBuffer heap-allocates a new contact buffer and adds it to the input Space's allocated buffers. Called by spacePushFreshContactBuffer to allocate a new ContactBufferHeader.
	Output:	spaceAllocContactBuffer returns a pointer to the allocated ContactBuffer as output, cast as a ContactBufferHeader pointer.
contactBuffer HeaderInit:	**Input:**	contactBufferHeaderInit accepts a ContactBufferHeader pointer, a Timestamp and another ContactBufferHeader pointer as input.
	Exceptions:	None.
	Transition:	contactBufferHeaderInit initializes the first input Contact-BufferHeader. It modifies its timestamp to the given Timestamp, its next header to be the next header of the second input ContactBufferHeader (or to the first input header if the second one is null), and its number of Contacts to zero. Called by spacePushFreshContactBuffer to initialize a ContactBufferHeader.
	Output:	contactBufferHeaderInit returns a pointer to the initialized ContactBufferHeader.

spacePush Contacts:	**Input:**	spacePushContacts accepts a Space pointer and an integer as input.
	Exceptions:	spacePushContacts may throw a BufferOverflow exception if the input integer exceeds the MAX_CONTACTS_PER_ARBITER. Called by spaceCollideShapes to process collisions.
	Transition:	spacePushContacts increases the number of Contacts of the input Space's contact buffer head by the given integer.
	Output:	None.
spacePop Contacts:	**Input:**	spacePopContacts accepts a Space pointer and an integer as inputs.
	Exceptions:	None.
	Transition:	spacePopContacts decrements the number of Contacts of the input Space's contact buffer head by the input integer. Called by spaceCollideShapes to process collisions.
	Output:	None.
queryReject:	**Input:**	queryReject accepts two Shape pointers as input.
	Exceptions:	None.
	Transition:	None.
	Output:	queryReject tests for collision conditions; in particular, it tests if the bounding boxes of both input Shapes overlap and if they belong to different Bodies. If all tests pass, the function returns true. Otherwise, it returns false. Called by spaceCollideShapes to validate collision.
spaceArbiter SetTrans:	**Input:**	spaceArbiterSetTrans accepts a pointer to an array of Shape pointers and a Space pointer.
	Exceptions:	spaceArbiterSetTrans may throw an InsufficientBufferSize exception if the input Space has no pooled Arbiters and the size of an Arbiter object exceeds the buffer size (BUFFER_BYTES).

Transition:	If the input Space has no pooled Arbiters, the function heap-allocates a new buffer, adds it to the Space's allocated buffers and adds new Arbiters to the Space's list of pooled Arbiters. It then obtains an Arbiter from the list and initializes it with the Shapes in the input array. Called by spaceCollideShapes to create Arbiters for colliding Shapes from pooled ones.
Output:	spaceArbiterSetTrans returns a pointer to the initialized Arbiter.

7 MIS of the Arbiter Module

7.1 Module Name: Arbiter

7.2 Uses

Rigid Body Module, Shape Module, Space Module, Control Module, Vector Module

7.3 Interface Syntax

7.3.1 Exported Constants

MAX_CONTACTS_PER_ARBITER: $\mathbb{Z}^+$
MAX_CONTACTS_PER_ARBITER := 2

7.3.2 Exported Data Types

ArbiterState: enum
ArbiterThread: struct
Contact: struct
CollisionInfo: struct
Arbiter: struct
ContactPointSet: struct

7.3.3 Exported Access Programs

Name	In	Out	Exceptions
arbiterInit	Arbiter*, Shape*, Shape*	Arbiter*	-
arbiterThreadForBody	Arbiter*, Body*	ArbiterThread	-
arbiterUnthread	Arbiter*	-	-
arbiterUpdate	Arbiter*, CollisionInfo*, Space*	-	-
arbiterPreStep	Arbiter*, double, double, double	-	-
arbiterApplyCachedImpulse	Arbiter*, double	-	-
arbiterApplyImpulse	Arbiter*	-	-
arbiterNext	Arbiter*, Body*	Arbiter*	-
arbiterGetRestitution	Arbiter*	double	-
arbiterGetFriction	Arbiter*	double	-
arbiterGetSurfaceVelocity	Arbiter*	Vector	-
arbiterGetCount	Arbiter*	int	-
arbiterGetNormal	Arbiter*	Vector	-
arbiterGetPointA	Arbiter*, int	Vector	ContactIndexOutOf Bounds
arbiterGetPointB	Arbiter*, int	Vector	ContactIndexOutOf Bounds
arbiterGetDepth	Arbiter*, int	double	ContactIndexOutOf Bounds
arbiterGetContactPointSet	Arbiter*	ContactPointSet	-
arbiterGetShapes	Arbiter*, Shape**, Shape**	-	
arbiterGetBodies	Arbiter*, Body**, Body**	-	
arbiterSetRestitution	Arbiter*, double	-	-
arbiterSetFriction	Arbiter*, double	-	-
arbiterSetSurfaceVelocity	Arbiter*, Vector	-	-
arbiterSetContactPointSet	Arbiter*, ContactPointSet*	-	ImmutableNum Contacts

arbiterIsFirstContact	Arbiter*	Boolean	-
arbiterIsRemoval	Arbiter*	Boolean	-
arbiterIgnore	Arbiter*	Boolean	-
arbiterTotalImpulse	Arbiter*	Vector	-
arbiterTotalKE	Arbiter*	double	-
arbiterCallWildcardBeginA	Arbiter*, Space*	Boolean	-
arbiterCallWildcardBeginB	Arbiter*, Space*	Boolean	-
arbiterCallWildcardPreSolveA	Arbiter*, Space*	Boolean	-
arbiterCallWildcardPreSolveB	Arbiter*, Space*	Boolean	-
arbiterCallWildcardPostSolveA	Arbiter*, Space*	-	-
arbiterCallWildcardPostSolveB	Arbiter*, Space*	-	-
arbiterCallWildcardSeparateA	Arbiter*, Space*	-	-
arbiterCallWildcardSeparateB	Arbiter*, Space*	-	-

7.4 Interface Semantics

7.4.1 State Variables

ArbiterState $\in$ {ARBITER_STATE_FIRST_COLLISION, ARBITER_STATE_NORMAL, ARBITER_STATE_IGNORE, ARBITER_STATE_CACHED, ARBITER_STATE_INVALIDATED}

ArbiterThread:

next: Arbiter*
prev: Arbiter*

Contact:

r1: Vector
r2: Vector
nMass: $\mathbb{R}$
tMass: $\mathbb{R}$
bounce: $\mathbb{R}$
jnAcc: $\mathbb{R}$
jtAcc: $\mathbb{R}$
jBias: $\mathbb{R}$
bias: $\mathbb{R}$
hash: HashValue

CollisionInfo:

a: Shape* (constant)
b: Shape* (constant)
id: CollisionID
normal: Vector
count: int
arr: Contact*

Arbiter:

elast: $\mathbb{R}$	bodyB: Body*	handler: CollisionHandler*
fric: $\mathbb{R}$	threadA: ArbiterThread	handlerA: CollisionHandler*
surfaceVel: Vector	threadB: ArbiterThread	handlerB: CollisionHandler*
a: Shape* (constant)	count: $\mathbb{Z}$	swapped: $\mathbb{B}$
b: Shape* (constant)	contacts: Contact*	stamp: Timestamp
bodyA: Body*	normal: Vector	state: ArbiterState

ContactPointSet:

count: $\mathbb{Z}$
normal: Vector
points: array({pointA: Vector, pointB: Vector, distance: $\mathbb{R}$}: struct)

7.4.2 Assumptions

All input pointers are assumed to be non-null.

7.4.3 Access Program Semantics

arbiterInit:	**Input:**	arbiterInit accepts an Arbiter pointer and two Shape pointers as input.
	Exceptions:	None.
	Transition:	arbiterInit initializes the input Arbiter. Its state is set to ARBITER_STATE_FIRST_COLLISION. Its Shapes and Bodies are set to the input Shapes and their associated Bodies, and all other fields are zero-initialized.
	Output:	arbiterInit returns a pointer to the initialized Arbiter as output.
arbiterThread ForBody:	**Input:**	arbiterThreadForBody accepts an Arbiter pointer and a Body pointer as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	arbiterThreadForBody returns the input Arbiter's ArbiterThread which corresponds to the input Body.

arbiterUnthread:	**Input:**	arbiterUnthread accepts an Arbiter pointer as input.
	Exceptions:	None.
	Transition:	arbiterUnthread calls unthreadHelper on the input Arbiter's Bodies to remove this Arbiter from the thread.
	Output:	None.
arbiterUpdate:	**Input:**	arbiterUpdate accepts an Arbiter pointer, a CollisionInfo pointer and a Space pointer as inputs.
	Exceptions:	None.
	Transition:	arbiterUpdate updates the Arbiter's state after a collision using the input CollisionInfo and Space. If the Arbiter had been cached, it changes the state to ARBITER_STATE_FIRST_COLLISION.
	Output:	None.
arbiterPreStep:	**Input:**	arbiterPreStep accepts an Arbiter pointer and three doubles as inputs.
	Exceptions:	None.
	Transition:	arbiterPreStep calculates the mass normal, mass tangent, bias velocity and bounce velocity for each of the input Arbiter's contacts, using the three input doubles which represent the timestep, collision slop and collision bias, respectively.
	Output:	None.
arbiterApply CachedImpulse:	**Input:**	arbiterApplyCachedImpulse accepts an Arbiter pointer and a double as inputs.
	Exceptions:	None.
	Transition:	arbiterApplyCachedImpulse applies the impulses stored in the input Arbiter's contacts to its Bodies, using the input double as a timestep coefficient.
	Output:	None.

arbiterApply Impulse:	**Input:**	arbiterApplyImpulse accepts an Arbiter pointer as input.
	Exceptions:	None.
	Transition:	arbiterApplyImpulse applies all impulses stored in the input Arbiter's contacts to its Bodies.
	Output:	None.
arbiterNext:	**Input:**	arbiterNext accepts an Arbiter pointer and a Body pointer as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	arbiterNext returns a pointer to the next Arbiter from the input Arbiter's ArbiterThread which corresponds to the input Body.
arbiterGet:	**Input:**	Each arbiterGet function accepts an Arbiter pointer as input. arbiterGetPointA, arbiterGetPointB and arbiterGetDepth accept an additional integer. arbiterGetShapes accepts two additional pointers to Shape pointers, while arbiterGetBodies accepts two additional pointers to Body pointers.
	Exceptions:	arbiterGetPointA, arbiterGetPointB and arbiterGetDepth may throw a ContactIndexOutOfBounds exception when the input integer exceeds the number of contact points for the input Arbiter.
	Transition:	arbiterGetContactPointSet initializes a new ContactPointSet for the input Arbiter using its array of Contacts. arbiterGetShapes and arbiterGetBodies retrieve the input Arbiter's Shapes and Bodies, respectively, and store them in the input pointers. All other arbiterGet functions make no transition.
	Output:	Each arbiterGet function, except for arbiterGetShapes and arbiterGetBodies, returns the value of their corresponding parameter.

arbiterSet:	**Input:**	Each arbiterSet function accepts an Arbiter pointer and their corresponding parameter as inputs. In particular, arbiterSetContactPointSet accepts a ContactPointSet pointer as input.
	Exceptions:	arbiterSetContactPointSet may throw an ImmutableNumContacts exception if the number of contact points in the input ContactPointSet differs from the current number of contact points of the input Arbiter.
	Transition:	Each arbiterSet function sets the value of their corresponding parameter with the input value. In particular, arbiterSetContactPointSet modifies the contents of the input Arbiter's array of Contacts according to the input ContactPointSet.
	Output:	None.
arbiterIs:	**Input:**	Each arbiterIs function accepts an Arbiter pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each arbiterIs function checks the state of the input Arbiter and returns a Boolean value according to the result.
arbiterIgnore:	**Input:**	arbiterIgnore accepts an Arbiter pointer as input.
	Exceptions:	None.
	Transition:	arbiterIgnore sets the state of the input Arbiter to ARBITER_STATE_IGNORE.
	Output:	arbiterIgnore always returns false.
arbiterTotal:	**Input:**	each arbiterTotal function accepts an Arbiter pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	Each arbiterTotal computes the total quantity of the corresponding parameter (impulse or kinetic energy) and returns the value as a double.

arbiterCall Wildcard:	**Input:**	Each arbiterCallWildcard function accepts an Arbiter pointer and a Space pointer as inputs.
	Exceptions:	None.
	Transition:	Each arbiterCallWildcard function calls the corresponding function from the input Arbiter's CollisionHandlers. The input arguments will be the input Arbiter, the input Space and the user data contained in the handler.
	Output:	Each arbiterCallWildcard returns the same value as that returned by the called function; PostSolve and Separate return a Boolean value, while Begin and PreSolve return nothing.

7.4.4 Local Functions

unthread Helper:	**Input:**	unthreadHelper accepts an Arbiter pointer and a Body pointer as inputs.
	Exceptions:	None.
	Transition:	unthreadHelper removes the input Arbiter from the ArbiterThread corresponding to the input Body, and may also remove the Arbiter from the Body. Called by arbiterUnthread.
	Output:	None.

8 MIS of the Control Module

8.1 Module Name: Chipmunk

8.2 Uses

This module only uses standard libraries.

8.3 Interface Syntax

8.3.1 Exported Constants

M_PI: $\mathbb{R}$
M_PI := 3.14159265358979323846

VOID_ERR is an empty macro definition.

PTR_ERR: void*
PTR_ERR := NULL

INT_ERR: $\mathbb{Z}$
INT_ERR := INT_MIN := -2147483648

DBL_ERR: $\mathbb{R}$
DBL_ERR := DBL_MIN := 1×10^{-37}

BUFFER_BYTES: $\mathbb{Z}^+$
BUFFER_BYTES := $32 \times 1024 = 32678$

WILDCARD_COLLISION_TYPE: CollisionType
WILDCARD_COLLISION_TYPE := 0

8.3.2 Exported Data Types

HashValue: pointer-compatible $\mathbb{Z}^+$
DataPointer: void*
CollisionType: pointer-compatible $\mathbb{Z}^+$
Timestamp: $\mathbb{Z}^+$
CollisionID: 32-bit $\mathbb{Z}^+$

8.3.3 Exported Access Programs

Name	In	Out	Exceptions
message	string, string, int, int, int, string	-	-
assertSoft	See note [1]	-	-
assertWarn	See note [1]	-	-
assertHard	See note [1]	-	-
fclamp	double, double, double	double	-
fclamp01	double	double	-
flerp	double, double, double	double	-
flerpconst	double, double, double	double	-

loopIndices	Vector*, int, int*, int*	-	-
convexHull	int, Vector*, Vector*, int*, double	int	-

[1] These assertions are defined as macros. They accept a Boolean expression, an error value (see 8.3.1), and an arbitrary number of arguments consisting of an error message and format parameters.

8.4 Interface Semantics

8.4.1 Access Program Semantics

message:

Input: message accepts two strings, three integers and a third string, followed by an arbitrary number of format arguments for this string.

Exceptions: None.

Transition: message will print a warning or error message to standard error, depending on the value of the second input integer (which should be non-zero for errors). It will then print the error message (third input string), with all the formatted data, to standard error, followed by the failed condition (first input string) and the source of the error, which includes the filename (second input string) and line number (first input integer).

Output: None.

assert:

Input: Each assert macro accepts a Boolean expression, an error value (see 8.3.1) and an arbitrary number of arguments including an error message and its format arguments.

Exceptions: None.

Transition: Each assert macro tests the input Boolean expression. If the test fails, each macro will print the input error message and its formatted message to standard error, and assertSoft and assertHard will abort the program immediately. In UNIT_TEST mode, each assert macro will return the input error value instead of aborting the program.

Output: None, normally. In UNIT_TEST mode, they may return the input error value.

fclamp:	**Input:**	Each fclamp function accepts a double as input. The regular fclamp accepts two additional doubles as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	fclamp restricts the first input double between the second and third input doubles, which are the min and max thresholds, respectively. It returns the first double if it falls within the specified range, the second double if it falls below the range, and the third double if it falls above the range. Similarly, fclamp01 restricts the input double between 0 and 1. Each fclamp function returns a double.
flerp:	**Input:**	Each flerp function accepts three doubles as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	flerp linearly interpolates between the first two input doubles for a percentage specified by the third input double. Similarly, flerpconst linearly interpolates between the first two doubles by no more than a constant specified in the third input double. Each flerp function returns a double.
loopIndices:	**Input:**	loopIndices accepts a pointer to a Vector array, an integer and two integer pointers as inputs.
	Exceptions:	None.
	Transition:	loopIndices iterates through the points contained in the input Vector array; the length of the array should be equal to the input integer. The function will determine the 'starting' (leftmost, bottommost) and 'ending' (rightmost, topmost) points in the array. It stores the indices of these starting and ending points in the first and second input integer pointers, respectively.
	Output:	None.
convexHull:	**Input:**	convexHull accepts an integer, two pointers to Vector arrays, an integer pointer and a double as inputs.
	Exceptions:	None.

	Transition:	convexHull calculates the hull size given by the set of points contained in the first input Vector array. If a valid integer pointer is provided, it will store the index of the first hull point.
	Output:	convexHull returns an integer for the number of points in the convex hull.

8.4.2 Local Functions

QHullPartition:	**Input:**	QHullPartition accepts a pointer to a Vector array, an integer, two Vectors and a double as inputs.
	Exceptions:	None.
	Transition:	QHullPartition partitions the set of points in the convex hull for reduction. Called by QHullReduce.
	Output:	QHullPartition returns an integer as output.
QHullReduce:	**Input:**	QHullReduce accepts a double, a pointer to a Vector array, an integer, three Vectors and another pointer to a Vector array.
	Exceptions:	None.
	Transition:	QHullReduce simplifies (shrinks) the convex hull by a certain tolerance, which is specified by the input double. Called as a helper for convexHull's divide-and-conquer approach.
	Output:	QHullReduce returns an integer as output.

9 MIS of the Vector Module

9.1 Module Name: Vector

9.2 Uses

This module only uses standard libraries.

9.3 Interface Syntax

9.3.1 Exported Constants

VECT_ERR, zeroVect: Vector
VECT_ERR := {INT_MAX, INT_MIN}
zeroVect := {0.0, 0.0}

9.3.2 Exported Data Types

Vector: struct

9.3.3 Exported Access Programs

Name	In	Out	Exceptions
vect	double, double	Vector	-
vectEqual	Vector, Vector	Boolean	-
vectAdd	Vector, Vector	Vector	-
vectSub	Vector, Vector	Vector	-
vectMult	Vector, double	Vector	-
vectNeg	Vector	Vector	-
vectDot	Vector, Vector	double	-
vectCross	Vector, Vector	double	-
vectPerp	Vector	Vector	-
vectRPerp	Vector	Vector	-
vectProject	Vector, Vector	Vector	-
vectForAngle	double	Vector	-
vectToAngle	Vector	double	-
vectRotate	Vector, Vector	Vector	-
vectUnrotate	Vector, Vector	Vector	-
vectLengthSq	Vector	double	-
vectLength	Vector	double	-
vectNormalize	Vector	Vector	-
vectClamp	Vector, double	Vector	-
vectLerp	Vector, Vector, double	Vector	-
vectDistSq	Vector, Vector	double	-

vectDist	Vector, Vector	double	-
vectNear	Vector, Vector, double	Boolean	-

9.4 Interface Semantics

9.4.1 State Variables

Vector:

x: $\mathbb{R}$
y: $\mathbb{R}$

9.4.2 Access Program Semantics

vect:

Input: vect accepts two doubles as input.
Exceptions: None.
Transition: None.
Output: vect returns a new Vector created from the input doubles.

vectEqual:

Input: vectEqual accepts two Vectors as input.
Exceptions: None.
Transition: None.
Output: vectEqual compares the values of the input Vectors and returns true if they are equal, and false otherwise.

vectAdd:

Input: vectAdd accepts two Vectors as input.
Exceptions: None.
Transition: None.
Output: vectAdd returns the sum of the input Vectors.

vectSub:

Input: vectSub accepts two Vectors as input.
Exceptions: None.
Transition: None.
Output: vectSub returns the difference of the input Vectors.

vectMult:	**Input:**	vectMult accepts a Vector and a double as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectMult returns the scalar multiple of the input Vector with the input double.
vectNeg:	**Input:**	vectNeg accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectNeg returns the negative of the input Vector.
vectDot:	**Input:**	vectDot accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectDot returns the dot product of the input Vectors.
vectCross:	**Input:**	vectCross accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectCross calculates the cross product of the input Vectors and returns the z-component of the product as a double.
vectPerp:	**Input:**	vectPerp accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectPerp rotates the input Vector by 90 degrees clockwise and returns the resultant Vector as output.
vectRPerp:	**Input:**	vectRPerp accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectRPerp rotates the input Vector by 90 degrees anti-clockwise and returns the resultant Vector as output.

vectProject:	**Input:**	vectProject accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectProject projects the first input Vector onto the second and returns the resultant Vector as output.
vectForAngle:	**Input:**	vectForAngle accepts a double as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectForAngle computes the Vector corresponding to the input angle (double), measured from the x-axis, and returns the result.
vectToAngle:	**Input:**	vectToAngle accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectToAngle calculates the angle between the input Vector and the x-axis and returns the result as a double.
vectRotate:	**Input:**	vectRotate accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectRotate rotates the first input Vector by the second using complex multiplication returns the resultant Vector as output.
vectUnrotate:	**Input:**	vectUnrotate accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectUnrotate is the inverse operation of vectRotate; it returns the original Vector before it was rotated by another Vector using vectRotate.
vectLength:	**Input:**	Each vectLength function accepts a Vector as input.
	Exceptions:	None.

	Transition:	None.
	Output:	vectLength and vectLengthSq calculates the regular and squared length of the input Vector, respectively, and returns the result as a double.
vectNormalize:	**Input:**	vectNormalize accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectNormalize converts the input Vector into a unit vector and returns the normalized Vector as output.
vectClamp:	**Input:**	vectClamp accepts a Vector and a double as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectClamp restricts the input Vector to a length specified by the input double. If the length of the input Vector is less than the input length, vectClamp returns the input Vector. Otherwise, it shrinks the Vector to the specified length and returns the resultant Vector.
vectLerp:	**Input:**	vectLerp accepts two Vectors and a double as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectLerp linearly interpolates between the two input Vectors for a percentage specified by the input double, and returns the new interpolated Vector as output.
vectDist:	**Input:**	Each vectDist function accepts two Vectors as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectDist and vectDistSq calculates the regular and squared distance, respectively, between the two input Vectors and returns the result as a double.
vectNear:	**Input:**	vectNear accepts two Vectors and a double as input.
	Exceptions:	None.

Transition: None.

Output: vectNear returns true if the distance between the input Vectors is less than the distance specified by the input double, and false otherwise.

10 MIS of the Bounding Box Module

10.1 Module Name: BB

10.2 Uses

Control Module, Vector Module

10.3 Interface Syntax

10.3.1 Exported Constants

BB_ERR: BB
BB_ERR := {INT_MAX, INT_MAX, INT_MIN, INT_MIN}

10.3.2 Exported Data Types

BB: struct

10.3.3 Exported Access Programs

Name	In	Out	Exceptions
BBNew	double, double, double, double	BB	-
BBNewForExtents	Vector, double, double	BB	NegativeHalf Dimensions
BBNewForCircle	Vector, double	BB	NegativeRadius
BBIntersects	BB, BB	Boolean	-
BBContainsBB	BB, BB	Boolean	-
BBContainsVect	BB, Vector	Boolean	-
BBMerge	BB, BB	BB	-
BBCenter	BB	Vector	-
BBArea	BB	double	-
BBMergedArea	BB, BB	double	-

BBClampVect	BB, Vector	Vector	-
BBWrapVect	BB, Vector	Vector	-
BBOffset	BB, Vector	BB	-

10.4 Interface Semantics

10.4.1 State Variables

BB:

left: $\mathbb{R}$ right: $\mathbb{R}$
bottom: $\mathbb{R}$ top: $\mathbb{R}$

10.4.2 Access Program Semantics

BBNew:

Input: BBNew accepts four doubles as input.

Exceptions: None.

Transition: BBNew allocates a new BB and initializes its left, bottom, right and top values with the input doubles, in that order.

Output: BBNew returns the allocated BB as output.

BBNewFor Extents:

Input: BBNewForExtents accepts a Vector and two doubles as inputs.

Exceptions: BBNewForExtents may raise a warning if the input doubles are negative.

Transition: BBNewForExtents creates a new BB centered on the input Vector. Its dimensions are calculated from the input doubles, which provide the half-width and half-height.

Output: BBNewForExtents returns the new BB as output.

BBNewFor Circle:

Input: BBNewForCircle accepts a Vector and a double as input.

Exceptions: BBNewForCircle may raise a warning if the input double is negative.

Transition: BBNewForCircle creates a new BB centered on the input Vector. Its radius is specified by the input double.

Output: BBNewForCircle returns the new BB as output.

BBIntersects:	**Input:**	BBIntersects accepts two BBs as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBIntersects returns true if the input BBs intersect on another, and false otherwise.
BBContainsBB:	**Input:**	BBContainsBB accepts two BBs as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBContainsBB returns true if the first input BB contains the second, and false otherwise.
BBContains Vect:	**Input:**	BBContainsVect accepts a BB and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBContainsVect returns true if the input Vector is within the bounds of the input BB, and false otherwise.
BBMerge:	**Input:**	BBMerge accepts two BBs as inputs.
	Exceptions:	None.
	Transition:	BBMerge creates a new BB containing the two input BBs.
	Output:	BBMerge returns the new BB as output.
BBCenter:	**Input:**	BBCenter accepts a BB as input.
	Exceptions:	None.
	Transition:	None.
	Output:	BBCenter returns the centroid of the input BB as a Vector.
BBArea:	**Input:**	BBArea accepts a BB as input.
	Exceptions:	None.
	Transition:	None.
	Output:	BBArea returns the area of the input BB as a double.

BBMergedArea:	**Input:**	BBMergedArea accepts two BBs as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBMergedArea returns the area of the region containing both input BBs as a double.
BBClampVect:	**Input:**	BBClampVect accepts a BB and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBClampVect restricts the input Vector to the dimensions of the input BB and returns the clamped Vector as output.
BBWrapVect:	**Input:**	BBWrapVect accepts a BB and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBWrapVect wraps the input Vector to the input BB and returns the wrapped Vector as output.
BBOffset:	**Input:**	BBOffset accepts a BB and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBOffset translates the input BB by the specified Vector and returns the shifted BB as output.

11 MIS of the Transform Matrix Module

11.1 Module Name: Transform

11.2 Uses

Vector Module, Bounding Box Module

11.3 Interface Syntax

11.3.1 Exported Constants

identity: Transform
identity := {1.0, 0.0, 0.0, 1.0, 0.0, 0.0}

11.3.2 Exported Data Types

Transform: struct

11.3.3 Exported Access Programs

Name	In	Out	Exceptions
transformNew	double, double, double, double, double, double	Transform	-
transformNewTranspose	double, double, double, double, double, double	Transform	-
transformInverse	Transform	Transform	-
transformMult	Transform, Transform	Transform	-
transformPoint	Transform, Vector	Vector	-
transformVect	Transform, Vector	Vector	-
transformBB	Transform, BB	BB	-
transformTranslate	Vector	Transform	-
transformScale	double, double	Transform	-
transformRotate	double	Transform	-
transformRigid	Vector, double	Transform	-
transformRigidInverse	Transform	Transform	-

11.4 Interface Semantics

11.4.1 State Variables

Transform:

a: $\mathbb{R}$ b: $\mathbb{R}$
c: $\mathbb{R}$ d: $\mathbb{R}$
tx: $\mathbb{R}$ ty: $\mathbb{R}$

11.4.2 Access Program Semantics

transformNew:	**Input:**	Each transformNew function accepts six doubles as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	transformNew and transformNewTranspose creates and returns a new Transform matrix from the input doubles in regular and transposed order, respectively.
transformInverse:	**Input:**	transformInverse accepts a Transform matrix as input.
	Exceptions:	None.
	Transition:	None.
	Output:	transformInverse calculates the inverse of the input Transform matrix and returns the result as output.
transformMult:	**Input:**	transformMult accepts two Transform matrices as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	transformMult multiplies the two input Transform matrices together and returns the result as output.
transformPoint:	**Input:**	transformPoint accepts a Transform matrix and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.

	Output:	transformPoint applies the affine transformation from the input Transform matrix to the input Vector and returns the resultant Vector.
transformVect:	**Input:**	transformVect accepts a Transform matrix and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	transformVect applies the linear transformation from the input Transform matrix to the input Vector and returns the resultant Vector.
transformBB:	**Input:**	transformBB accepts a Transform matrix and a BB as inputs.
	Exceptions:	None.
	Transition:	transformBB calculates the half-dimensions of the input BB, applies the input Transform matrix to calculate the transformed dimensions, computes the new transformed half-dimensions, and creates a new BB from the new half-dimensions. The center of this new BB is obtained by applying the input Transform matrix to the centroid of the old BB.
	Output:	transformBB returns the new, transformed BB as output.
transform Translate:	**Input:**	transformTranslate accepts a Vector as input.
	Exceptions:	None.
	Transition:	transformTranslate creates a translation matrix from the input Vector.
	Output:	transformTranslate returns the new Transform matrix as output.
transformScale:	**Input:**	transformScale accepts two doubles as inputs.
	Exceptions:	None.
	Transition:	transformScale creates a scaling matrix from the input doubles, which represent the horizontal and vertical scale factors, respectively.

	Output:	transformScale returns the new Transform matrix as output.
transformRotate:	**Input:**	transformRotate accepts a double as input.
	Exceptions:	None.
	Transition:	transformRotate calculates a Vector from the angle specified by the input double and creates a rotation matrix from the Vector.
	Output:	transformRotate returns the new Transform matrix as output.
transformRigid:	**Input:**	transformRigid accepts a Vector and a double as inputs.
	Exceptions:	None.
	Transition:	transformRigid calculates a Vector from the angle specified by the input double and creates a rigid transformation matrix from the input parameters, using the computed Vector for the rotation components and the input Vector for the translation components.
	Output:	transformRigid returns the new Transform matrix as output.
transformRigid Inverse:	**Input:**	transformRigidInverse accepts a Transform matrix as input.
	Exceptions:	None.
	Transition:	None.
	Output:	transformRigidInverse returns the inverse of a rigid Transform matrix.

12 MIS of the Spatial Index Module

12.1 Module Name: SpatialIndex

12.2 Uses

Control Module, Vector Module, Bounding Box Module, Linked Data Structure Module

12.3 Interface Syntax

12.3.1 Exported Data Types

SpatialIndex: struct
SpatialIndexClass: struct
DynamicToStaticContext: struct
SpatialIndexBBFunc : void* → BB
SpatialIndexIteratorFunc : void* × void* → void
SpatialIndexQueryFunc : void* × void* × CollisionID × void* → CollisionID
SpatialIndexDestroyImpl : SpatialIndex* → void
SpatialIndexCountImpl : SpatialIndex* → $\mathbb{Z}$
SpatialIndexEachImpl : SpatialIndex* × SpatialIndexIteratorFunc × void* → void
SpatialIndexContainsImpl : SpatialIndex* × void* × HashValue → $\mathbb{B}$
SpatialIndexInsertImpl : SpatialIndex* × void* × HashValue → void
SpatialIndexRemoveImpl : SpatialIndex* × void* × HashValue → void
SpatialIndexReindexImpl : SpatialIndex* → void
SpatialIndexReindexObjectImpl : SpatialIndex* × void* × HashValue → void
SpatialIndexReindexQueryImpl : SpatialIndex* × SpatialIndexQueryFunc × void* → void
SpatialIndexQueryImpl : SpatialIndex* × void* × BB × SpatialIndexQueryFunc × void* → void

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
spatialIndexInit	SpatialIndex*, SpatialIndex-Class*, SpatialIndexBB-Func, SpatialIndex*	SpatialIndex*	AttachedStaticIndex
spatialIndexFree	SpatialIndex*	-	-
spatialIndexCollideStatic	SpatialIndex*, SpatialIndex*, SpatialIndex-QueryFunc, void*	-	-
spatialIndexDestroy	SpatialIndex*	-	-
spatialIndexCount	SpatialIndex*	int	-

spatialIndexEach	SpatialIndex*, SpatialIndexIteratorFunc, void*	-	-
spatialIndexContains	SpatialIndex*, void*, HashValue	Boolean	-
spatialIndexInsert	SpatialIndex*, void*, HashValue	-	-
spatialIndexRemove	SpatialIndex*, void*, HashValue	-	-
spatialIndexReindex	SpatialIndex*	-	-
spatialIndexReindexObject	SpatialIndex*, void*, HashValue	-	-
spatialIndexQuery	SpatialIndex*, void*, BB, SpatialIndexQueryFunc, void*	-	-
spatialIndexReindexQuery	SpatialIndex*, SpatialIndexQueryFunc, void*	-	-

12.4 Interface Semantics

12.4.1 State Variables

SpatialIndex:

klass: SpatialIndexClass*
bbfunc: SpatialIndexBBFunc
staticIndex: SpatialIndex*
dynamicIndex: SpatialIndex*

SpatialIndexClass:

destroy: SpatialIndexDestroyImpl
count: SpatialIndexCountImpl
each: SpatialIndexEachImpl
contains: SpatialIndexContainsImpl
insert: SpatialIndexInsertImpl
remove: SpatialIndexRemoveImpl
reindex: SpatialIndexReindexImpl
reindexObject: SpatialIndexReindexObjectImpl
reindexQuery: SpatialIndexReindexQueryImpl
query: SpatialIndexQueryImpl

DynamicToStaticContext:

bbfunc: SpatialIndexBBFunc	queryFunc: SpatialIndexQueryFunc
staticIndex: SpatialIndex*	data: void*

12.4.2 Assumptions

spatialIndexInit is called before any other access program. All input pointers are also assumed to be non-null.

12.4.3 Access Program Semantics

spatialIndex Init:	**Input:**	spatialIndexInit accepts a SpatialIndex pointer, a SpatialIndexClass pointer, a SpatialIndexBBFunc function pointer, and another SpatialIndex pointer as inputs.
	Exceptions:	spatialIndexInit may throw an AttachedStaticIndex exception if the last input SpatialIndex pointer for the staticIndex is already associated to another dynamicIndex.
	Transition:	spatialIndexInit initializes the first input SpatialIndex with the input parameters and zero-initializes other fields.
	Output:	spatialIndexInit returns a pointer to the initialized SpatialIndex as output.
spatialIndex Free:	**Input:**	spatialIndexFree accepts a SpatialIndex pointer as input.
	Exceptions:	None.
	Transition:	spatialIndexFree frees the input SpatialIndex.
	Output:	None.
spatialIndex CollideStatic:	**Input:**	spatialIndexCollideStatic accepts two SpatialIndex pointers, a SpatialIndexQueryFunc function pointer, and a void pointer as inputs.
	Exceptions:	None.
	Transition:	If the input static index (second SpatialIndex) is valid and non-empty, the function creates a new DynamicToStaticContext using the input parameters, sets its bbfunc to the bbfunc of the input dynamic index (first SpatialIndex).

		Afterwards, it iterates through the dynamic index using dynamicToStaticIter and the new context.
	Output:	None.
spatialIndex Destroy:	**Input:**	spatialIndexDestroy accepts a SpatialIndex pointer as input.
	Exceptions:	None.
	Transition:	spatialIndexDestroy calls the internal destroying function from the input SpatialIndex's class with the index itself.
	Output:	None.
spatialIndex Count:	**Input:**	spatialIndexCount accepts a SpatialIndex pointer as input.
	Exceptions:	None.
	Transition:	spatialIndexCount calls the internal counting function from the input SpatialIndex's class with the index itself.
	Output:	spatialIndexCount returns the integer result of the counting function as output.
spatialIndex Each:	**Input:**	spatialIndexEach accepts a SpatialIndex pointer, a SpatialIndexIteratorFunc function pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	spatialIndexEach calls the internal iterator function from the SpatialIndex's class with the input parameters.
	Output:	None.
spatialIndex Contains:	**Input:**	spatialIndexContains accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	spatialIndexContains calls the internal existence-checking function from the input SpatialIndex's class with the input parameters.

	Output:	spatialIndexContains returns true if the input index contains the object in the input void pointer, and false otherwise.
spatialIndex Insert:	**Input:**	spatialIndexInsert accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	spatialIndexInsert calls the internal insertion function from the input SpatialIndex's class with the input parameters.
	Output:	None.
spatialIndex Remove:	**Input:**	spatialIndexRemove accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	spatialIndexRemove calls the internal removal function from the input SpatialIndex's class with the input parameters.
	Output:	None.
spatialIndex Reindex:	**Input:**	spatialIndexReindex accepts a SpatialIndex pointer as input.
	Exceptions:	None.
	Transition:	spatialIndexReindex calls the internal reindexing function from the input SpatialIndex's class with the index itself.
	Output:	None.
spatialIndex ReindexObject:	**Input:**	spatialIndexReindexObject accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	spatialIndexReindexObject calls the internal object-reindexing function from the input SpatialIndex's class with the input parameters.
	Output:	None.

spatialIndex Query:	**Input:**	spatialIndexQuery accepts a SpatialIndex pointer, a void pointer, a BB and a SpatialIndexQueryFunc function pointer.
	Exceptions:	None.
	Transition:	spatialIndexQuery calls the internal querying function from the input SpatialIndex's class with the input parameters.
	Output:	None.
spatialIndex ReindexQuery:	**Input:**	spatialIndexReindexQuery accepts a SpatialIndex pointer, a SpatialIndexQueryFunc function pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	spatialIndexReindexQuery calls the internal query-based reindexing function from the input SpatialIndex's class with the input parameters.
	Output:	None.

12.4.4 Local Functions

dynamicTo StaticIter:	**Input:**	dynamicToStaticIter accepts a void pointer and a DynamicToStaticContext pointer as inputs.
	Exceptions:	None.
	Transition:	dynamicToStaticIter queries the index by calling spatialIndexQuery with the input void pointer and the fields of the input DynamicToStaticContext.
	Output:	None.

13 MIS of the Collision Solver Module

13.1 Module Name: Collision

13.2 Uses

Rigid Body Module, Shape Module, Arbiter Module, Control Module, Vector Module, Bounding Box Module

13.3 Interface Syntax

13.3.1 Exported Constants

POINTS_ERR: ClosestPoints
POINTS_ERR := {VECT_ERR, VECT_ERR, DBL_MIN, UINT32_MAX}

MAX_GJK_ITERATIONS, MAX_EPA_ITERATIONS, WARN_GJK_ITERATIONS, WARN_-EPA_ITERATIONS: $\mathbb{Z}$
MAX_GJK_ITERATIONS := 30
MAX_EPA_ITERATIONS := 30
WARN_GJK_ITERATIONS := 20
WARN_EPA_ITERATIONS := 20

13.3.2 Exported Data Types

SupportPoint: struct
MinkowskiPoint: struct
SupportContext: struct
EdgePoint: struct
Edge: struct
ClosestPoints: struct
CollisionFunc : Shape* $\times$ Shape* $\times$ CollisionInfo* $\rightarrow$ void
SupportPointFunc : Shape* $\times$ Vector $\rightarrow$ SupportPoint

13.3.3 Exported Access Programs

Name	In	Out	Exceptions
relative_velocity	Body*, Body*, Vector, Vector	Vector	-
normal_relative_velocity	Body*, Body*, Vector, Vector, Vector	double	-
apply_impulse	Body*, Vector, Vector	-	-
apply_impulses	Body*, Body*, Vector, Vector, Vector	-	-
apply_bias_impulse	Body*, Vector, Vector	-	-

apply_bias_impulses	Body*, Body*, Vector, Vector, Vector	-	-
k_scalar_body	Body*, Vector, Vector	double	-
k_scalar	Body*, Body*, Vector, Vector, Vector	double	UnsolvableCollision
collide	Shape*, Shape*, CollisionID, Contact*	CollisionInfo	-
shapesCollide	Shape*, Shape*	ContactPointSet	-

13.4 Interface Semantics

13.4.1 State Variables

SupportPoint:

p: Vector
index: CollisionID

MinkowskiPoint:

a: Vector
b: Vector
ab: Vector
id: CollisionID

SupportContext:

shape1: Shape*
shape2: Shape*
func1: SupportPointFunc
func2: SupportPointFunc

EdgePoint:

p: Vector
hash: HashValue

Edge:

a: EdgePoint
b: EdgePoint
radius: $\mathbb{R}$
normal: Vector

ClosestPoints:

a: Vector	n: Vector	d: $\mathbb{R}$
b: Vector		id: CollisionID

13.4.2 Access Program Semantics

relative_velocity:	**Input:**	relative_velocity accepts two Body pointers and two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	relative_velocity calculates the relative velocity of the second input Body relative to the first input Body with the input parameters and returns the result as a Vector.
normal_relative_velocity:	**Input:**	normal_relative_velocity accepts two Body pointers and three Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	normal_relative_velocity calculates the dot product of the relative velocity between the two input Bodies and the normal (third input Vector) and returns the result as a double.
apply_impulse:	**Input:**	apply_impulse accepts a Body pointer and two Vectors as inputs.
	Exceptions:	None.
	Transition:	apply_impulse recalculates the input Body's linear and angular velocity using the impulse (first input Vector) and point of application (second input Vector).
	Output:	None.
apply_impulses:	**Input:**	apply_impulses accepts two Body pointers and three Vectors as inputs.
	Exceptions:	None.

	Transition:	apply_impulses applies the input impulse (third input Vector) to the two input Bodies, in opposite directions, to recalculate their linear and angular velocities, using their points of application (first and second input Vectors).
	Output:	None.
apply_bias_impulse:	**Input:**	apply_bias_impulse accepts a Body pointer and two Vectors as inputs.
	Exceptions:	None.
	Transition:	apply_bias_impulse recalculates the input Body's linear and angular bias velocities using the impulse (first input Vector) and point of application (second input Vector).
	Output:	None.
apply_bias_impulses:	**Input:**	apply_bias_impulses accepts two Body pointers and three Vectors as inputs.
	Exceptions:	None.
	Transition:	apply_bias_impulses applies the input impulse (third input Vector) to the two input Bodies, in opposite directions, to recalculate their linear and angular bias velocities, using their points of application (first and second input Vectors).
	Output:	None.
k_scalar_body:	**Input:**	k_scalar_body accepts a Body pointer and two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	k_scalar_body first calculates the cross product of the two input Vectors. Then, it computes the product of the inverse momentum of the input Body and the squared cross product of the input Vectors. Finally, it calculates the sum of this quantity and the Body's inverse mass, and returns the final result as a double.
k_scalar:	**Input:**	k_scalar accepts two Body pointers and three Vectors as inputs.

	Exceptions:	k_scalar may throw an UnsolvableCollision exception if the calculated value is equal to zero.
	Transition:	None.
	Output:	k_scalar calculates k_scalar_body for the first input Body with the first and last input Vector, and for the second input Body with the second and last input Vector. It then calculates the sum of these results and returns the above sum as a double.
collide:	**Input:**	collide accepts two Shape pointers, a CollisionID and a Contact pointer as inputs.
	Exceptions:	None.
	Transition:	collide creates a new CollisionInfo structure with the input parameters and other fields zero-initialized. The function will then reorder the structure's Shape types as necessary, and apply the appropriate collision function from CollisionFuncs to it.
	Output:	collide returns the new CollisionInfo structure as output.
shapesCollide:	**Input:**	shapesCollide accepts two Shape pointers as inputs.
	Exceptions:	None.
	Transition:	shapesCollide declares a new Contact array and generates a CollisionInfo structure for the input Shapes using the collide function and the Contact array, modifying the array in the process. Next, it declares a new ContactPointSet structure for the collision and sets the number of points and normal accordingly. Finally, the function will iterate through the Contact array to set the points for the ContactPointSet.
	Output:	shapesCollide returns the new ContactPointSet as output.

13.4.3 Local Constants

BuiltinCollisionFuncs: array(CollisionFunc)
BuiltinCollisionFuncs := {CircleToCircle, CollisionError, CollisionError, CircleToSegment, SegmentToSegment, CollisionError, CircleToPoly, SegmentToPoly, PolyToPoly}
CollisionFuncs := BuiltinCollisionFuncs

13.4.4 Local Functions

collisionInfo PushContact:

Input: collisionInfoPushContact accepts a CollisionInfo pointer, two Vectors and a HashValue as inputs.

Exceptions: collisionInfoPushContact may throw a CollisionContactOverflow exception when the number of Contacts of the input CollisionInfo exceeds MAX_CONTACTS_PER_ARBITER.

Transition: collisionInfoPushContact pushes a new Contact structure into the input CollisionInfo's Contacts array with the other input parameters and updates its number of Contacts accordingly. Called by the ShapeToShape collision functions to add new contact points and by closestPoints in the collision functions for SegmentShapes and PolyShapes.

Output: None.

SupportPoint:

Input: Each SupportPoint function accepts a Shape pointer of the Shape type corresponding to the function's prefix and a Vector as inputs.

Exceptions: None.

Transition: None.

Output: Each SupportPoint creates a new SupportPoint with the input Shape's transformed center (CircleShapes), endpoint (SegmentShape) or vertex (PolyShape), with the appropriate index of the point as its CollisionID. Each corresponding function is used by the appropriate ShapeToShape function in generating the SupportPointContext to be passed to GJK.

support:

Input: support accepts a SupportContext pointer and a Vector as inputs.

Exceptions: None.

Transition: support calculates the maximal point on the Minkowski difference of two shapes along a particular axis. It generates two SupportPoints using the SupportPointFunc functions and Shapes contained in the input SupportContext and the input Vector, and creates a new MinkowskiPoint with these SupportPoints. Used in the calculations of GJK and EPA.

	Output:	support returns the new MinkowskiPoint as output.
supportEdgeFor:	**Input:**	Each supportEdgeFor function accepts a Shape pointer of the corresponding Shape type and a Vector as inputs.
	Exceptions:	None.
	Transition:	Each supportEdgeFor function computes the dot products of the input Shape's vertices (for PolyShapes) or normal (for SegmentShapes) with the input Vector to calculate a support edge for the input Shape, which is an edge of a SegmentShape or PolyShape that is in contact with another Shape. Called by some ShapeToShape functions to determine contact points for SegmentShapes and PolyShapes.
	Output:	Each supportEdgeFor function generates a new Edge structure containing information about the calculated support edge and returns it as output.
closestT:	**Input:**	closestT accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	closestT finds the closest $\mathbf{p}(t)$ to the origin $(0, 0)$, where $\mathbf{p}(t) = \frac{a(1-t)+b(1+t)}{2}$, a and b are the two input Vectors and $t \in [-1, 1]$. The function clamps the result to this interval. Used for the computation of closest points in closestPointsNew.
	Output:	closestT returns a double as output.
lerpT:	**Input:**	lerpT accepts two Vectors and a double as inputs.
	Exceptions:	None.
	Transition:	lerpT functions similarly to vectLerp, except the parameter t, the last input double, is constrained to the interval $[-1, 1]$. Used for the computation of closest points in closestPointsNew.
	Output:	lerpT returns a Vector as output.
closestPoints New:	**Input:**	closestPointsNew accepts two MinkowskiPoint structures as inputs.
	Exceptions:	None.

Transition: closestPointsNew finds the closest edge to the origin (0, 0) on the Minkowski difference of two Shapes, which is obtained by using closestT and lerpT with the input MinkowskiPoints. This is used to calculate the closest points on the surface of two Shapes, as well as the distance and the minimum separating axis between them. The function then generates a new ClosestPoints structure using the calculated data and the concatenated IDs of the input MinkowskiPoints. Used to compute closest points in EPA and GJK.

Output: closestPointsNew returns the new ClosestPoints as output.

EPA:

Input: EPA accepts a SupportContext pointer, and three MinkowskiPoint structures as inputs.

Exceptions: EPA may throw a SameVertices exception when the EPA vertices are the same. It may also raise HighIterWarning when the number of iterations reaches the WARN_EPA_ITERATIONS threshold.

Transition: EPA recursively finds the closest points on the surface of two overlapping Shapes using the EPA (Expanding Polytope Algorithm). The function initializes a convex hull array of vertices and each recursion adds a point to the hull until the function obtains the closest points on the surfaces of the Shapes.

Output: EPA generates a new ClosestPoints structure containing information about the computed closest points and returns it as output.

GJK:

Input: GJK accepts a SupportContext pointer and a CollisionID pointer as inputs.

Exceptions: GJK may raise a HighIterWarning when the number of iterations reaches the WARN_GJK_ITERATIONS threshold, or WARN_EPA_ITERATIONS when EPA needs to be called.

Transition: GJK recursively finds the closest points between two shapes using the (Gilbert-Johnson-Keerthi) algorithm. If the collision Shapes are found to overlap at some iteration of the algorithm, the function will then execute EPA to find the closest points.

	Output:	GJK generates a new ClosestPoints structure containing information about the computed closest points and returns it as output.
contactPoints:	**Input:**	contactPoints accepts two Edge structures, a ClosestPoints structure and a CollisionInfo pointer as inputs.
	Exceptions:	None.
	Transition:	contactPoints finds contact point pairs on the surfaces of the input support Edges and pushes a new Contact structure into the input CollisionInfo's Contacts array. This is used in ShapeToShape functions involving SegmentShapes and PolyShapes (except for CircleToPoly).
	Output:	None.
ShapeToShape:	**Input:**	Each ShapeToShape function accept two pointers to the corresponding Shape types and a CollisionInfo pointer as inputs.
	Exceptions:	None.
	Transition:	Each ShapeToShape function calls GJK to find the ClosestPoints for the two input Shapes and uses it to check if the current distance between the two Shapes is less than the minimum collision distance (usually determined by the sum of the Shapes' radii). If so, the function pushes a new Contact structure containing information about the Shapes' contact points into the Contacts array of the input CollisionInfo. These functions are stored in the exported CollisionFuncs array, and the appropriate function will be called by collide.
	Output:	None.
CollisionError:	**Input:**	CollisionError accepts two Shape pointers and a CollisionInfo pointer as inputs.
	Exceptions:	CollisionError throws an eponymous exception when the types of the input Shapes are not in sorted order.
	Transition:	CollisionError throws an exception and aborts the program. This function is stored in the exported CollisionFuncs array and called by collide when the colliding Shape types are not in order.

Output: None.

14 MIS of the Sequence Data Structure Module

14.1 Module Name: Array

14.2 Uses

This module only uses standard libraries.

14.3 Interface Syntax

14.3.1 Exported Data Types

Array: struct

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
arrayNew	int	Array*	-
arrayFree	Array*	-	-
arrayPush	Array*, void*	-	-
arrayPop	Array*	void*	EmptyArray
arrayDeleteObj	Array*, void*	-	-
arrayContains	Array*, void*	Boolean	-
arrayFreeEach	Array*, void* $\rightarrow$ void	-	-

14.4 Interface Semantics

14.4.1 State Variables

Array:

num: $\mathbb{Z}$
max: $\mathbb{Z}$
arr: void**

14.4.2 State Invariant

Array.num $\leq$ Array.max

14.4.3 Assumptions

arrayNew is called before any other access program, and all input pointers are assumed to be non-null.

14.4.4 Access Program Semantics

arrayNew:	**Input:**	arrayNew accepts an integer as input.
	Exceptions:	None.
	Transition:	arrayNew heap-allocates a new Array object. It sets the Array's length and maximum length to the input integer (unless the input is zero, in which case the default maximum is 4), and heap-allocates a maximum-length void pointer array for the internal array.
	Output:	arrayNew returns the newly-created Array as output.
arrayFree:	**Input:**	arrayFree accepts an Array pointer as input.
	Exceptions:	None.
	Transition:	arrayFree frees the internal array of the input Array, and then frees the Array itself.
	Output:	None.
arrayPush:	**Input:**	arrayPush accepts an Array pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	arrayPush inserts the specified element into the input Array and increments the number of elements accordingly. If the Array is at capacity, the function will double the maximum length and resize the internal array accordingly.
	Output:	None.
arrayPop:	**Input:**	arrayPop accepts an Array pointer as input.
	Exceptions:	arrayPop may throw an EmptyArray exception if the user attempts to pop items from an empty Array.
	Transition:	arrayPop will remove the last element of the input Array and decrements the number of elements accordingly.

	Output:	arrayPop returns the retrieved void pointer as output.
arrayDeleteObj:	**Input:**	arrayDeleteObj accepts an Array pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	arrayDeleteObj deletes the specified element (void pointer) from the input Array and decrements the number of elements accordingly.
	Output:	None.
arrayContains:	**Input:**	arrayContains accepts an Array pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	arrayContains iterates through the input Array and returns true if the Array contains the input void pointer, and false otherwise.
arrayFreeEach:	**Input:**	arrayFreeEach accepts an Array pointer and a pointer to a freeing function that accepts a void pointer and returns nothing (void* $\rightarrow$ void).
	Exceptions:	None.
	Transition:	arrayFreeEach iterates through the internal array of the input Array and applies the input function to each element.
	Output:	None.

15 MIS of the Linked Data Structure Module

15.1 Module Name: BBTree

15.2 Uses

Control Module, Vector Module, Bounding Box Module, Spatial Index Module, Sequence Data Structure Module, Associative Data Structure Module

15.3 Interface Syntax

15.3.1 Exported Constants

NODE_ERR: Node
NODE_ERR := {NULL, BB_ERR, NULL}

PAIR_ERR: Pair
PAIR_ERR = {{NULL, NULL, NULL}, {NULL, NULL, NULL}, UINT32_MAX}

15.3.2 Exported Data Types

BBTree: struct
Node: struct
Thread: struct
Pair: struct
MarkContext: struct
EachContext: struct
BBTreeVelocityFunc : void* → Vector

15.3.3 Exported Access Programs

Name	In	Out	Exceptions
BBTreeAlloc	-	BBTree*	-
BBTreeInit	BBTree*, SpatialIndexBBFunc, SpatialIndex*	SpatialIndex*	-
BBTreeSetVelocityFunc	SpatialIndex*, BBTreeVelocityFunc	-	NotBBTreeWarn
BBTreeNew	SpatialIndexBBFunc, SpatialIndex*	SpatialIndex*	-
BBTreeDestroy	BBTree*	-	-
BBTreeCount	BBTree*	int	-
BBTreeEach	BBTree*, SpatialIndexIteratorFunc, void*	-	-
BBTreeInsert	BBTree*, void*, HashValue	-	-
BBTreeRemove	BBTree*, void*, HashValue	-	-

BBTreeContains	BBTree*, void*, HashValue	-	-
BBTreeReindexQuery	BBTree*, SpatialIndexQueryFunc, void*	-	-
BBTreeReindex	BBTree*	-	-
BBTreeReindexObject	BBTree*, void*, HashValue	-	-

15.4 Interface Semantics

15.4.1 State Variables

BBTree:

spatialIndex: SpatialIndex
velocityFunc: BBTreeVelocityFunc
leaves: HashSet*
root: Node*
pooledNodes: Node*
pooledPairs: Pair*
allocatedBuffers: Array*
stamp: Timestamp

Node:

obj: void*
bb: BB
parent: Node*
node: union

Node.node.children:

a: Node*
b: Node*

Node.node.leaf:

stamp: Timestamp
pairs: Pair*

Thread:

prev: Pair*
leaf: Node*
next: Pair*

Pair:

a: Thread
b: Thread
id: CollisionID

MarkContext:

tree: BBTree*
staticRoot: Node*
func: SpatialIndexQueryFunc
data: void*

EachContext:

func: SpatialIndexIteratorFunc
data: void*

15.4.2 Assumptions

BBTreeAlloc or BBTreeNew is called before any other access program, and all input pointers are assumed to be non-null.

15.4.3 Access Program Semantics

BBTreeAlloc:	**Input:**	None.
	Exceptions:	None.
	Transition:	BBTreeAlloc allocates a new BBTree from the heap.
	Output:	BBTreeAlloc returns a pointer to the allocated BBTree.
BBTreeInit:	**Input:**	BBTreeInit accepts a BBTree pointer, a SpatialIndexBBFunc function pointer and a SpatialIndex pointer as inputs.
	Exceptions:	None.
	Transition:	BBTreeInit initializes the input BBTree as a SpatialIndex, using the BBTree SpatialIndexClass klass and the input parameters. All internal data structures are created accordingly and other variables are zero-initialized.
	Output:	BBTreeInit returns a general SpatialIndex pointer to the initialized BBTree.
BBTreeSet VelocityFunc:	**Input:**	BBTreeSetVelocityFunc accepts a SpatialIndex pointer and a BBTreeVelocityFunc function pointer as inputs.

	Exceptions:	BBTreeSetVelocityFunc may raise a NotBBTree warning if the input SpatialIndex is not a BBTree.
	Transition:	BBTreeSetVelocityFunc sets the input BBTree SpatialIndex's internal velocity function to the provided function.
	Output:	None.
BBTreeNew:	**Input:**	BBTreeNew accepts a SpatialIndexBBFunc function pointer and a SpatialIndex pointer as inputs.
	Exceptions:	None.
	Transition:	BBTreeNew allocates a new BBTree from the heap and initializes it.
	Output:	BBTreeNew returns a pointer to the initialized BBTree.
BBTreeDestroy:	**Input:**	BBTreeDestroy accepts a BBTree pointer as input.
	Exceptions:	None.
	Transition:	BBTreeDestroy frees the dynamically-allocated structures of the input BBTree and all of its elements, namely its leaves and allocated buffers.
	Output:	None.
BBTreeCount:	**Input:**	BBTreeCount accepts a BBTree pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	BBTreeCount counts the number of leaves contained in the input Tree and returns the result as an integer.
BBTreeEach:	**Input:**	BBTreeEach accepts a BBTree pointer, a SpatialIndexIteratorFunc function pointer, and a void pointer as inputs.
	Exceptions:	None.
	Transition:	BBTreeEach creates a new EachContext structure with the input function and data (void pointer), and iterates through the input BBTree's leaves using the hash set iterator and the context structure.
	Output:	None.

BBTreeInsert:	**Input:**	BBTreeInsert accepts a BBTree pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	BBTreeInsert inserts a new Node with the input object (void pointer) and HashValue into the input BBTree. The function will update the Node's timestamp to the master tree's timestamp, add appropriate Pairs for the Node and update the timestamp of the input tree.
	Output:	None.
BBTreeRemove:	**Input:**	BBTreeRemove accepts a BBTree pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	BBTreeRemove deletes the Node corresponding to the input object (void pointer) and HashValue from the input BBTree, clears the Pairs for that Node and recycles the empty Node.
	Output:	None.
BBTreeContains:	**Input:**	BBTreeContains accepts a BBTree pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	BBTreeContains searches the leaves of the input BBTree for the input object (void pointer) and HashValue. It returns true if a valid Node is found; otherwise, it returns false.
BBTreeReindex Query:	**Input:**	BBTreeReindexQuery accepts a BBTree pointer, a SpatialIndexQueryFunc function pointer, and a void pointer as inputs.
	Exceptions:	None.
	Transition:	If the input BBTree does not have a valid root, the function returns immediately. Otherwise, it will update the tree's leaves, attempt to obtain the root of the tree's static index, create a new MarkContext structure with the root and the input parameters, and mark the tree with this context.

		If the static index does not have a valid root, the function calls spatialIndexCollideStatic with the index and the input parameters. Finally, the function updates the timestamp of the tree.
	Output:	None.
BBTreeReindex:	**Input:**	BBTreeReindex accepts a BBTree pointer as input.
	Exceptions:	None.
	Transition:	BBTreeReindex calls BBTreeReindexQuery with the input BBTree, voidQueryFunc and a null pointer.
	Output:	None.
BBTreeReindex Object:	**Input:**	BBTreeReindexObject accepts a BBTree pointer, a void pointer and a HashValue as inputs.
	Exceptions:	None.
	Transition:	The function will attempt to find the Node corresponding to the input HashValue and object (void pointer). If found, it will attempt to update the Node, and following success, add Pairs for the Node. The tree's timestamp will be updated accordingly at the end of the function.
	Output:	None.

15.4.4 Local Constants

klass: SpatialIndexClass
klass := {BBTreeDestroy, BBTreeCount, BBTreeEach, BBTreeContains, BBTreeInsert, BBTreeRemove, BBTreeReindex, BBTreeReindexObject, BBTreeReindexQuery}

15.4.5 Local Functions

Recycle:	**Input:**	Each Recycle function accepts a BBTree pointer and a pointer to the corresponding object as inputs.
	Exceptions:	None.

Transition: pairRecycle retrieves the master tree (its index of dynamic bodies) of the input BBTree and recycles the input Pair to the tree's pooled Pairs, while nodeRecycle removes the input Node from the input BBTree and recycles it to the tree's pooled Nodes. Used in functions involving deletion (pairsClear, subtreeRemove, nodeReplaceChild, BBTreeRemove) and recycling fresh objects from the buffer (FromPool).

Output: None.

FromPool:

Input: pairFromPool accepts a BBTree pointer as input.

Exceptions: Each FromPool function may throw an InsufficientBufferSize exception if the size of the corresponding object exceeds the buffer size (BUFFER_BYTES).

Transition: pairFromPool retrieves the master tree of the input BBTree, while nodeFromPool uses the input BBTree. Each function retrieves a new object the tree's pooled objects. If there is none, the function allocates a new object buffer, adds it to the tree's allocated buffers, and adds all the new objects to the pool except the first one, which is returned. Used in pairInsert and Node constructors nodeNew, leafNew.

Output: pairFromPool returns a pointer to the retrieved Pair as output.

pairsClear:

Input: pairsClear accepts a Node pointer and a BBTree pointer as inputs.

Exceptions: None.

Transition: pairsClear removes all Pairs associated with the input Node, unlinks all Threads associated with each Pair and recycles the Pairs. Used in leafUpdate and BBTreeRemove.

Output: None.

pairInsert:

Input: pairInsert accepts two Node pointers and a BBTree pointer as inputs.

Exceptions: None.

	Transition:	pairInsert obtains a pooled Pair from the input BBTree, creates a new Pair with the input Nodes and inserts the Pair by linking it with the Threads associated with the Nodes. Used in markLeafQuery.
	Output:	None.
nodeNew:	**Input:**	nodeNew accepts a BBTree pointer and two Node pointers as inputs.
	Exceptions:	None.
	Transition:	nodeNew retrieves a pooled Node and initializes it. The function sets this Node's children to the two input Nodes, its BB to the merged BBs of the two Nodes, and all other values to null. Used for inserting new Nodes in subtreeInsert.
	Output:	nodeNew returns the initialized Node as output.
nodeReplace Child:	**Input:**	nodeReplaceChild accepts three Node pointers and a BBTree pointer as inputs.
	Exceptions:	nodeReplaceChild may throw a LeafError exception if the user attempts to replace a child of a leaf Node, or an InvalidChild exception if the child Node (second input Node) is not a child of the parent Node (first input Node).
	Transition:	nodeReplaceChild replaces the child (second input Node) of the parent Node (first input Node) with the third input Node, and updates the BBs of all parents of the parent Node. Used in the deletion algorithm of subtreeRemove.
	Output:	None.
subtreeInsert:	**Input:**	subtreeInsert accepts two Node pointers and a BBTree pointer as inputs.
	Exceptions:	None.
	Transition:	subtreeInsert inserts the second input Node into the subtree originating from the first input Node, and recalculates the BB of this root. Main insertion subroutine used in leafUpdate and BBTreeInsert.
	Output:	subtreeInsert returns a pointer to the resultant subtree as output.

subtreeRemove:	**Input:**	subtreeRemove accepts two Node pointers and a BBTree pointer as inputs.
	Exceptions:	None.
	Transition:	subtreeRemove deletes the second input Node and its parent from the subtree originating from the first input Node. Main deletion subroutine used in leafUpdate and BBTreeRemove.
	Output:	subtreeRemove returns a pointer to the resultant subtree as output.
markLeafQuery:	**Input:**	markLeafQuery accepts two Node pointers, a Boolean value and a MarkContext pointer as inputs.
	Exceptions:	None.
	Transition:	markLeafQuery only makes transitions if the BBs of the two input Nodes intersect. In that case, the function will check if the first input Node is a leaf. If not, the function will recursively search through the left and right subtrees of the first input Node. Otherwise, and if the Node is a left child (the input Boolean value is true), a new Pair will be added with the two Nodes. If the Node is a right child (the input Boolean value is false), a Pair will be added if the Node was updated more recently than the first Node. Finally, markLeafQuery will call the function in the input MarkContext with the objects of both Nodes, the ID zero, and the data (void pointer) in the context. This querying function is used in markLeaf.
	Output:	None.
markLeaf:	**Input:**	markLeaf accepts a Node pointer and a MarkContext pointer as inputs.
	Exceptions:	None.
	Transition:	markLeaf checks if first input Node was last updated at the same time as the master tree. If this is true, and if the input MarkContext has a valid static root, the function will call markLeafQuery with the root, the input Node, a value of false and the MarkContext.

		Afterwards, the function will iteratively move up each Node in the tree and call markLeafQuery on their siblings. If the timestamps are different, the function will instead iterate through the Pairs of the input Node. For each Pair, it will check if the Node is a leaf of the B-thread, and if so, call the function in the given MarkContext with the object of the leaf in the A-thread, the object of the Node, the ID of the Pair, and the data (void pointer) in the context. In this case, the function then traverses through the Pairs in the B-thread; otherwise, it traverses through the Pairs in the A-thread. Main subroutine used to mark tree leaves in markSubtree.
	Output:	None.
markSubtree:	**Input:**	markSubtree accepts a Node pointer and a MarkContext pointer as inputs.
	Exceptions:	None.
	Transition:	markSubtree checks if the input Node is a leaf. If so, it will mark the Node. Otherwise, markSubtree will recursively find and mark its children. This recursive subroutine is used to mark nodes through queries in BBTreeReindexQuery.
	Output:	None.
leafNew:	**Input:**	leafNew accepts a BBTree pointer, a void pointer and a BB as inputs.
	Exceptions:	None.
	Transition:	leafNew retrieves a pooled Node from the input Tree and initializes the Node with the input object (void pointer) and BB. All other variables are zero-initialized. Used for inserting new leaves in BBTreeInsert.
	Output:	leafNew returns a pointer to the initialized leaf Node as output.
leafUpdate:	**Input:**	Each leafUpdate function accepts a Node pointer and a BBTree pointer as inputs.
	Exceptions:	None.

	Transition:	leafUpdate obtains the BB corresponding to the object of the first input Node. If the Node's BB contains this BB, the function will update the Node; it sets the Node's BB to its object's BB, updates the Node's position in the tree, clear its Pairs, and updates its timestamp. leafUpdateWrap is simply a void-returning wrapper for this function. These are used in reindexing operations BBTreeReindexQuery and BBTreeReindexObject which require the leaves to be updated.
	Output:	If the Node is updated, leafUpdate returns true. Otherwise, it returns false. leafUpdateWrap calls this function, but discards the Boolean output.
leafAddPairs:	**Input:**	leafAddPairs accepts a Node pointer and a BBTree pointer as inputs.
	Exceptions:	None.
	Transition:	leafAddPairs attempts to retrieve the master tree. If the tree is valid and has a valid root, the function creates a new MarkContext structure with the tree and call markLeafQuery on the root, the input Node, a value of true and the context structure. Otherwise, it will obtain the root of the index of static bodies, create a new MarkContext structure with the input Tree, the root, and voidQueryFunc, and mark the input Node using this context. Used to add Pairs to leaves in BBTreeInsert and BBTreeReindexObject.
	Output:	None.

16 MIS of the Associative Data Structure Module

16.1 Module Name: HashSet

16.2 Uses

Control Module, Sequence Data Structure Module

16.3 Interface Syntax

16.3.1 Exported Constants

HASH_COEF: $\mathbb{Z}^+$
HASH_COEF := 3344921057

BIN_ERR: HashSetBin
BIN_ERR := {NULL, UINTPTR_MAX, NULL}

16.3.2 Exported Data Types

HashSetBin: struct
HashSet: struct
HashSetEqlFunc : void* $\times$ void* $\rightarrow \mathbb{B}$
HashSetTransFunc : void* $\times$ void* $\rightarrow$ void
HashSetIteratorFunc : void* $\times$ void* $\rightarrow$ void
HashSetFilterFunc : void* $\times$ void* $\rightarrow \mathbb{B}$

16.3.3 Exported Access Programs

Name	In	Out	Exceptions
HASH_PAIR	HashValue, HashValue	HashValue	-
hashSetNew	int, HashSetEqlFunc	HashSet*	-
hashSetSetDefaultValue	HashSet*, void*	-	-
hashSetFree	HashSet*	-	-
hashSetCount	HashSet*	int	-
hashSetInsert	HashSet*, HashValue, void*, HashSetTransFunc, void*	void*	-
hashSetRemove	HashSet*, HashValue, void*	void*	-
hashSetFind	HashSet*, HashValue, void*	void*	-
hashSetEach	HashSet*, HashSetIteratorFunc, void*	-	-

hashSetFilter	HashSet*, HashSetFilterFunc, void*	-	-

16.4 Interface Semantics

16.4.1 State Variables

HashSetBin:

elt: void*
hash: HashValue
next: HashSetBin*

HashSet:

entries: $\mathbb{Z}^+$ | defaultVal: void* | pooledBins: HashSetBin*
size: $\mathbb{Z}^+$ | table: HashSetBin** | allocatedBuffers: Array*
eql: HashSetEqlFunc

16.4.2 State Invariant

HashSet.entries $\leq$ HashSet.size

16.4.3 Assumptions

hashSetNew is called before any other access programs, and all input pointers are assumed to be non-null.

16.4.4 Access Program Semantics

HASH_PAIR:

Input: HASH_PAIR is a macro that accepts two HashValues as inputs.

Exceptions: None.

Transition: None.

Output: HASH_PAIR calculates a new HashValue from the pair of input HashValues and returns it as output.

hashSetNew:

Input: hashSetNew accepts an integer and a HashSetEqlFunc function pointer as inputs.

Exceptions: None.

	Transition:	hashSetNew heap-allocates a new HashSet, where the size is the next prime number greater than the input integer. The HashSet's internal equality function is set to the input function, and other variables are zero-initialized. Finally, a new Array is created for the HashSet's allocated buffers, and HashSetBins are allocated for the internal hash table.
	Output:	hashSetNew returns a pointer to the newly-created HashSet.
hashSetSet DefaultValue:	**Input:**	hashSetSetDefaultValue accepts a HashSet pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	hashSetSetDefaultValue sets the input HashSet's default value variable to the input void pointer.
	Output:	None.
hashSetFree:	**Input:**	hashSetFree accepts a HashSet pointer as input.
	Exceptions:	None.
	Transition:	hashSetFree frees the internal hash table of the input HashSet, frees its allocated buffers, and finally frees the HashSet itself.
	Output:	None.
hashSetCount:	**Input:**	hashSetCount accepts a HashSet pointer as input.
	Exceptions:	None.
	Transition:	None.
	Output:	hashSetCount returns the number of hash table entries contained in the input HashSet as an integer.
hashSetInsert:	**Input:**	hashSetInsert accepts a HashSet pointer, a HashValue, a void pointer, a HashSetTransFunc function pointer and another void pointer as inputs.
	Exceptions:	None.

Transition: hashSetInsert inserts the input element (first void pointer) into the input HashSet and increments its number of entries accordingly, if the element does not already exist. The element is placed in a bin and transformed if the appropriate function and information (second void pointer) are provided. If the HashSet is at capacity, it will be resized accordingly.

Output: hashSetInsert returns a void pointer to the inserted element as output.

hashSetRemove: **Input:** hashSetRemove accepts a HashSet pointer, a HashValue and a void pointer as inputs.

Exceptions: None.

Transition: hashSetRemove deletes the input element (void pointer) from the input HashSet and decrements its number of entries accordingly, if it exists. The bin containing the element is recycled in the process.

Output: hashSetRemove returns a void pointer to the removed element as output. If the element does not exist, it returns a null value.

hashSetFind: **Input:** hashSetFind accepts a HashSet pointer, a HashValue and a void pointer as inputs.

Exceptions: None.

Transition: None.

Output: hashSetFind searches through the input HashSet and attempts to find the input element (void pointer). If the element is found, hashSetFind returns a void pointer to the element as output. Otherwise, it returns the HashSet's default value.

hashSetEach: **Input:** hashSetEach accepts a HashSet pointer, a HashSetIteratorFunc function pointer and a void pointer as inputs.

Exceptions: None.

Transition: hashSetEach iterates through the entries of the input HashSet and calls the input function on each element with the input void pointer.

	Output:	None.
hashSetFilter:	**Input:**	hashSetFilter accepts a HashSet pointer, a HashSetFilterFunc function pointer and a void pointer as inputs.
	Exceptions:	None.
	Transition:	hashSetFilter iterates through the entries of the input HashSet and removes all entries for which the input filtering function returns false. For each removed element, the bin containing the entry is recycled. The number of entries is updated accordingly.
	Output:	None.

16.4.5 Local Constants

primes: array($\mathbb{Z}$)
primes := {5, 13, 23, 47, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317, 196613, 393241, 786433, 1572869, 3145739, 6291469, 12582917, 25165843, 50331653, 100663319, 201326611, 402653189, 805306457, 1610612741, 0}

16.4.6 Local Functions

next_prime:	**Input:**	next_prime accepts an integer as input.
	Exceptions:	next_prime may throw an IllegalSize exception if the input integer is greater than 1610612741.
	Transition:	next_prime iterates through the primes array and finds the nearest prime that is greater than the input integer. Used to determine new size in hashSetResize.
	Output:	next_prime returns the next greatest prime as output.
hashSetResize:	**Input:**	hashSetResize accepts a HashSet pointer as input.
	Exceptions:	None.
	Transition:	hashSetResize allocates a new hash table for the input HashSet, approximately double its current size. Each element is rehashed and reinserted into the new table, and the old table is freed. The capacity and number of entries are updated accordingly.
	Output:	None.

recycleBin:	**Input:**	recycleBin accepts a HashSet pointer and a HashSetBin pointer as inputs.
	Exceptions:	None.
	Transition:	recycleBin deletes the element of the input HashSetBin and adds it to the input HashSet's pooled bins. Used for deletion in hashSetRemove and hashSetFilter, and to recycle bins fresh from the buffer in getUnusedBin.
	Output:	None.
getUnusedBin:	**Input:**	getUnusedBin accepts a HashSet pointer as input.
	Exceptions:	getUnusedBin may throw a InsufficientBufferSize exception if the size of a HashSetBin object exceeds the buffer size. (BUFFER_BYTES).
	Transition:	getUnusedBin retrieves the first unused bin from the input HashSet's pooled bins. If there are no pooled bins, the function allocates a new HashSetBin buffer, adds it to the HashSet's allocated buffers, and adds all the new bins to the pool except for the first one, which is returned. Used for insertion in hashSetInsert.
	Output:	getUnusedBin returns a HashSetBin pointer to the retrieved bin.

17 Appendix

Table 1: Possible Exceptions

Exception Name	**Error Message**
AttachedBody	"You have already added this body to another space. You cannot add it to a second."
AttachedShape	"You have already added this shape to another space. You cannot add it to a second."
AttachedStaticBody	"Internal Error: Changing the designated static body while the old one still had shapes attached."
AttachedStaticIndex	"This static index is already associated with a dynamic index."
BodyNotFound	"Cannot remove a body that was not added to the space. (Removed twice maybe?)"
BufferOverflow	"Internal Error: Contact buffer overflow!"
CollisionContactOverflow	"Internal Error: Tried to push too many contacts."
CollisionError	"Internal Error: Shape types are not sorted."
ContactIndexOutOfBounds	"Index error: The specified contact index is invalid for this arbiter."
DuplicateBody	"You have already added this body to this space. You must not add it a second time."
DuplicateShape	"You have already added this shape to this space. You must not add it a second time."
EmptyArray	"Unable to pop items from an empty array!"
HighIterWarning	One of: "High EPA iterations: #," "High GJK iterations: #," "High GJK->EPA iterations: #," where # is the number of iterations.
IllegalBody	One of the above messages, in addition to: "Body's position is invalid.", "Body's velocity is invalid," "Body's force is invalid," "Body's angle is invalid," "Body's angular velocity is invalid," "Body's torque is invalid."
IllegalSize	"Tried to resize a hash table to a size greater than 1610612741."
ImmutableNumContacts	"The number of contact points cannot be changed."
IndexOutOfBounds	"Index out of range."
InfiniteMass	"Mass must be positive and finite."
InsufficientBufferSize	"Internal Error: Buffer size too small."
InvalidChild	"Internal Error: Node is not a child of parent."

InvalidIter	“Iterations must be positive and non-zero.”
LeafError	“Internal Error: Cannot replace child of a leaf.”
MainStaticBody	“Cannot remove the designated static body for the space.”
NaNMass	“Body’s mass is NaN.”
NaNMoment	“Body’s moment is NaN”.
NegativeElasticity	“Elasticity must be a positive quantity.”
NegativeFriction	“Friction must be a positive quantity.”
NegativeHalfDimensions	“Half-dimensions should be nonnegative.”
NegativeMass	“Body’s mass is negative.”
NegativeMoment	“Body’s moment is negative.”
NegativeRadius	“Radius should be nonnegative.”
NotBBTreeWarn	“Ignoring BBTreeSetVelocityFunc() call to non-tree spatial index.”
NotCircleShape	“Shape is not a circle shape.”
NotSegmentShape	“Shape is not a segment shape.”
NotPolyShape	“Shape is not a poly shape.”
SameVertices	“Internal Error: EPA vertices are the same (#1 and #2),” where #1 and #2 are the indices of the vertices.
ShapeNotFound	“Cannot remove a shape that was not added to the space. (Removed twice maybe?)”
SpaceLocked	One of: “This operation cannot be done safely during a call to spaceStep() or during a query. Put these calls into a post-step callback,” “You cannot manually reindex objects while the space is locked. Wait until the current query or step is complete.”
SpaceLockUnderflow	“Internal Error: Space lock underflow.”
StaticBodyMass	“You cannot set the mass of static bodies.”
UnsolvableCollision	“Unsolvable collision or constraint.”