Module Interface Specification for Chipmunk2D Game Physics Library

Luthfi Mawarid

August 16, 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, signed and unsigned integers, double-precision floating-point numbers (doubles), and pointers. 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.
Integer	$\mathbb{Z}$	A number without a fractional component in (-∞, ∞).
Unsigned integer	$\mathbb{Z}^+$	A number without a fractional component in $[0, \infty)$.
Double	$\mathbb{R}$	Any number in (-∞, ∞).
Pointer	$T*$	A reference to an object of data type T.

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

Data Type	**Notation**	**Description**
Array	array(T)	A list of a given 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.
Enumeration	enum	A data type containing named, constant values.

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, go to Section 3 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 $\vee$ NaNMoment $\vee$ NegativeMass $\vee$ NegativeMoment $\vee$ 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 $\vee$ NegativeMass $\vee$ 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 $\in [0, \infty)$
- $|$Body.pos.x$| \in \mathbb{R} \wedge |$Body.pos.y$| \in \mathbb{R}$
- $|$Body.vel.x$| \in \mathbb{R} \wedge |$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 inputted pointers are also assumed to not be null.

4.4.4 Access Program Semantics

bodyAlloc:

Input: bodyAlloc does not accept any input.

Exceptions: There are no potential exceptions for bodyAlloc.

Transition: bodyAlloc heap-allocates a new Body object.

Output: bodyAlloc returns a pointer to the allocated Body object.

bodyInit:

Input: bodyInit accepts a Body pointer and two double values as inputs.

Exceptions: There are no potential exceptions for bodyInit.

Transition: bodyInit allocates a new Body and initializes its mass and moment with the inputted 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 an exception when the user provides NaN or negative values for input, or when the user provides an infinite value for the first inputted double.

Transition: newBody will allocate a new Body, initialize it with the inputted and default values, and set it to a dynamic body.

Output: newBody returns a pointer to the new Body.

newStaticBody:

Input: newStaticBody does not accept any input.

Exceptions: There are no potential exceptions for newStaticBody.

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: There are no potential exceptions for bodyDestroy.

Transition: bodyDestroy frees the inputted Body object.

Output: bodyDestroy does not return any value.

bodyGet:

Input: Each bodyGet function accepts a Body pointer as input.

Exceptions: There are no potential exceptions for each bodyGet function.

Transition: Each bodyGet function makes no transitions.

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.

Transitions: Each bodySet function will modify the state of their corresponding parameter to the inputted 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: Each bodySet function does not return a value.

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: bodyAccumulateMassFromShapes does not return a value.

bodyAddShape:

Input: bodyAddShape accepts a Body pointer and a Shape pointer as inputs.

Exceptions: There are no potential exceptions for bodyAddShape.

Transition: The function will add the inputted Shape to the inputted Body's list of Shapes and recalculate the Body's mass accordingly.

Output: bodyAddShape does not return a value.

bodyRemove Shape:

Input: bodyRemoveShape accepts a Body pointer and a Shape pointer as inputs.

Exceptions: There are no potential exceptions for bodyRemoveShape.

Transition: The function will remove the inputted Shape from the Body's list of Shapes and, if the Body is dynamic, recalculate its mass accordingly.

Output: bodyRemoveShape does not return a value.

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: bodyUpdatePosition does not return any value.

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 inputted gravity Vector, forces and torques applied, and the inputted 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: bodyUpdateVelocity does not return any value.

bodyKinetic Energy:

Input: bodyKineticEnergy accepts a Body pointer as input.

Exceptions: There are no potential exceptions for bodyKineticEnergy.

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: There are no potential exceptions for each bodyEach function.

Transition: Each bodyEach function will iterate through the Body's Shapes or Arbiters, depending on the function's corresponding parameter, and apply the inputted function to each object in the list, using the data (void pointer) from the third inputted value.

Output: Each bodyEach function does not return any value.

4.4.5 Local Functions

vectAssertNaN:

Input: vectAssertNaN accepts a Vector and a string as inputs.

Exceptions: vectAssertNaN throws an exception if the inputted Vector has NaN values for its fields.

Transition: vectAssertNaN checks if the fields of the inputted Vector are valid numbers. If this test fails, it prints the inputted string as an error message.

Output: vectAssertNaN does not return any value.

vectAssertInfinite:

Input: vectAssertInfinite accepts a Vector and a string as inputs.

Exceptions: vectAssertInfinite throws an exception if the inputted Vector has infinite values for its fields.

Transition: vectAssertInfinite checks if the fields of the inputted Vector are finite. If this test fails, it prints the inputted string as an error message.

Output: vectAssertInfinite does not return any value.

vectAssertSane:

Input: vectAssertSane accepts a Vector and a string as inputs.

Exceptions: vectAssertSane throws an exception if the inputted Vector has NaN or infinite values for its fields.

Transition: vectAssertSane checks if the fields of the inputted Vector are valid and finite double values. If this test fails, it prints the inputted string as an error message.

Output: vectAssertSane does not return any value.

assertSaneBody:

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 inputted Body satisfies all state invariants, and prints various error messages depending on the first invariant found to be violated.

Output: assertSaneBody does not return any value.

bodySet Transform:

Input: bodySetTransform accepts a Body pointer, a Vector, and a double value as inputs.

Exceptions: There are no potential exceptions for bodySetTransform.

Transition: bodySetTransform mutates the inputted Body's transformation matrix using the inputted position Vector and a rotation vector converted from the given angle (last inputted value).

Output: bodySetTransform does not return any value.

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* × Transform → BB
ShapeDestroyImpl : Shape* → 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: ℝ
moment: ℝ

com: Vector
area: ℝ

ShapeClass:

type: ShapeType
cacheData: ShapeCacheDataImpl

destroy: ShapeDestroyImpl

Shape:

klass: ShapeClass*
space: Space*
body: Body*
massInfo: ShapeMassInfo
bb: BB
elast: ℝ

fric: 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 inputted 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: There are no potential exceptions for shapeInit.

Transition: shapeInit initializes the inputted Shape. It sets the Shape's class, body and mass information to the inputted 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: There are no potential exceptions for shapeDestroy.

Transition: shapeDestroy frees the inputted Shape object.

Output: shapeDestroy does not return any value.

shapeGet:

Input: Each shapeGet function accepts a Shape pointer as input.

Exceptions: There are no potential exceptions for each shapeGet function.

Transition: Each shapeGet makes no transitions.

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: Each shapeSet function does not return any value.

shapeCacheBB:

Input: shapeCacheBB accepts a Shape pointer as input.

Exceptions: There are no potential exceptions for shapeCacheBB.

Transition: shapeCacheBB updates the inputted 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: There are no potential exceptions for shapeUpdate.

Transition: shapeUpdate will call the cacheData function in the inputted 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 world coordinates.

5.8.2 Assumptions

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

5.8.3 Access Program Semantics

circleShapeAlloc:

Input: circleShapeAlloc does not accept any input.

Exceptions: There are no potential exceptions for circleShapeAlloc.

Transition: circleShapeAlloc heap-allocates a new CircleShape object.

Output: circleShapeAlloc returns a pointer to the allocated CircleShape as output.

circleShapeInit:

Input: circleShapeInit accepts a CircleShape pointer, a Body pointer, a double and a Vector as inputs.

Exceptions: There are no potential exceptions for circleShapeInit.

Transition: circleShapeInit initializes the inputted CircleShape. It sets the radius to the inputted double value, the center to the inputted Vector, and then initializes the rest using shapeInit and the inputted 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: There are no potential exceptions for circleShapeNew.

Transition: circleShapeNew allocates and initializes a new CircleShape object using the inputted 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 an exception if the inputted Shape pointer is not of the CircleShape class.

Transition: Each circleShapeGet function makes no transition.

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 an exception if the inputted 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 inputted value, updates the mass information of the Shape and recalculates the mass of its associated Body.

Output: Each circleShapeSet function does not return any value.

momentForCircle: **Input:** momentForCircle accepts three doubles for mass, inner radius and outer radius, and a Vector as inputs.

Exceptions: There are no potential exceptions for momentForCircle.

Transition: momentForCircle makes no transitions.

Output: momentForCircle returns the calculated moment from the inputted parameters as a double.

areaForCircle: **Input:** areaForCircle accepts two double values for the inner radius and outer radius as inputs.

Exceptions: There are no potential exceptions for areaForCircle.

Transition: areaForCircle makes no transitions.

Output: areaForCircle returns the calculated area from the inputted 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: There are no potential exceptions for circleShapeCacheData.

Transition: circleShapeCacheData updates the transformed center of the inputted CircleShape using the inputted Transform matrix and generates a new BB with the CircleShape's properties.

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: There are no potential exceptions for circleShapeMassInfo.

Transition: circleShapeMassInfo stack-allocates a new ShapeMassInfo structure using the inputted values.

Output: circleShapeMassInfo returns the allocated ShapeMassInfo structure.

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 world coordinates.

5.12.2 Assumptions

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

5.12.3 Access Program Semantics

segmentShapeAlloc:

Input: segmentShapeAlloc does not accept any input.

Exceptions: There are no potential exceptions for segmentShapeAlloc.

Transition: segmentShapeAlloc heap-allocates a new SegmentShape object.

Output: segmentShapeAlloc returns a pointer to the allocated SegmentShape as output.

segmentShapeInit:

Input: segmentShapeInit accepts a SegmentShape pointer, a Body pointer, two Vectors and a double as inputs.

Exceptions: There are no potential exceptions for segmentShapeInit.

Transition: segmentShapeInit initializes the inputted SegmentShape. It sets the endpoints to the given Vectors, the radius to the given double, and initializes the rest with shapeInit and the inputted Body.

Output: segmentShapeInit returns a pointer to the initialized SegmentShape.

segmentShapeNew:

Input: segmentShapeNew accepts a Body pointer, two Vectors and a double as inputs.

Exceptions: There are no potential exceptions for segmentShapeNew.

Transition: segmentShapeNew allocates and initializes a new SegmentShape object using the inputted parameters.

Output: segmentShapeNew returns a pointer to the new SegmentShape.

segmentShapeGet:

Input: Each segmentShapeGet function accepts a Shape pointer as input.

Exceptions: Each segmentShapeGet function may throw an exception if the inputted Shape pointer is not of the SegmentShape class.

Transition: Each segmentShapeGet function makes no transitions.

Output: Each segmentShapeGet function returns the value of their corresponding parameter.

segmentShapeSet:

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 an exception if the inputted Shape pointer is not of the SegmentShape class. segmentShapeSetEndpoints and segmentShapeSetRadius may throw an 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 inputted value. In addition, segmentShapeSetEndpoints and segmentShapeSetRadius update the mass information of the Shape and recalculate the mass of the associated Body.

Output: Each segmentShapeSet function does not return any value.

momentForSegment:

Input: momentForSegment accepts a double for mass, two Vectors for endpoints, and another double for radius as inputs.

Exceptions: There are no potential exceptions for momentForSegment.

Transition: momentForSegment makes no transitions.

Output: momentForSegment returns the calculated moment from the inputted parameters as a double.

areaForSegment:

Input: areaForSegment accepts two Vectors for endpoints and a double for radius as inputs.

Exceptions: There are no potential exceptions for areaForSegment.

Transition: areaForSegment makes no transitions.

Output: areaForSegment returns the calculated area from the inputted 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: There are no potential exceptions for segmentShapeCacheData.

Transition: segmentShapeCacheData updates the transformed endpoints and normal for the inputted SegmentShape using the inputted Transform matrix and generates a new BB with the SegmentShape's properties.

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: There are no potential exceptions for segmentShapeMassInfo.

Transition: segmentShapeMassInfo stack-allocates a new ShapeMassInfo structure using the inputted values.

Output: segmentShapeMassInfo returns the allocated ShapeMassInfo structure.

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*	-

polyShapeInitRaw	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 $\vee$ IndexOutOfBounds
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: ℝ
count: 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 inputted pointers are also assumed to be non-null.

5.16.3 Access Program Semantics

polyShapeAlloc:

Input: polyShapeAlloc does not accept any input.

Exceptions: There are no potential exceptions for polyShapeAlloc.

Transition: polyShapeAlloc heap-allocates a new PolyShape object.

Output: polyShapeAlloc returns a pointer to the allocated PolyShape 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: There are no potential exceptions for polyShapeInit.

Transition: polyShapeInit allocates an array of Vectors of length equivalent to the inputted integer, transforms each Vector in the inputted array using the given matrix and places them in the allocated array. The function then calculates the size of the convex hull containing the points in the array and initializes the inputted PolyShape using this array, the hull size and the remaining parameters.

Output: polyShapeInit returns a pointer to the initialized PolyShape.

polyShapeInitRaw:

Input: polyShapeInitRaw accepts a PolyShape pointer, a Body pointer, an integer, a pointer to a Vector array and a double as inputs.

Exceptions: There are no potential exceptions for polyShapeInitRaw.

Transition: polyShapeInitRaw initializes the inputted PolyShape using shapeInit and the inputted parameters, sets its vertices to the given array and integer (which represents the length of the array), and sets its radius to the inputted 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: There are no potential exceptions for boxShapeInit.

Transition: boxShapeInit calculates values for half-width and half-height using the last two inputted doubles as width and height, respectively. It then initializes the inputted 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: There are no potential exceptions for boxShapeInit2.

Transition: boxShapeInit2 creates a Vector array containing the vertices of the box, determined from the inputted BB. It then initializes the inputted 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: There are no potential exceptions for each polyShapeNew function.

Transition: Each polyShapeNew function allocates and initializes a new PolyShape object using the inputted 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: There are no potential exceptions for each boxShapeNew function.

Transition: Each boxShapeNew function allocates and initializes a new PolyShape object as a box using the inputted 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 an exception if the inputted Shape pointer is not of the PolyShape class. polyShapeGetVert may also throw an exception if the inputted integer is greater than or equal to the number of vertices of the inputted Shape.

Transition: Each polyShapeGet function makes no transition.

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. In particular, each polyShapeSetVerts function accepts an integer (for the number of vertices) and a pointer to a Vector array (holding the vertices) as inputs. polyShapeSetVerts (not Raw) also accepts an additional Transform matrix.

Exceptions: Each polyShapeSet function may throw an exception if the inputted Shape pointer is not of the PolyShape class. Each polyShapeSetVerts function may throw an 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 inputted value. In addition, each polyShapeSetVerts function updates the mass information of the Shape and recalculates the mass of the associated Body.

Output: Each polyShapeSet function does not return any value.

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: There are no potential exceptions for momentForPoly.

Transition: momentForPoly makes no transitions.

Output: momentForPoly returns the calculated moment from the inputted 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: There are no potential exceptions for areaForPoly.

Transition: areaForPoly makes no transitions.

Output: areaForPoly returns the calculated area from the inputted 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: There are no potential exceptions for centroidForPoly.

Transition: centroidForPoly makes no transitions.

Output: centroidForPoly returns the calculated centroid from the inputted 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: There are no potential exceptions for polyShapeDestroy.

Transition: polyShapeDestroy frees the inputted PolyShape and its associated array of vertices.

Output: polyShapeDestroy does not return any value.

polyShape CacheData:

Input: polyShapeCacheData accepts a PolyShape pointer and a Transform matrix as inputs.

Exceptions: There are no potential exceptions for polyShapeCacheData.

Transition: polyShapeCacheData transforms each vertex of the inputted PolyShape using the inputted matrix, calculates the extreme points of the shape, and updates the PolyShape's BB to a new BB generated from the calculated extremes.

Output: polyShapeCacheData returns the new BB as output.

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: There are no potential exceptions for polyShapeMassInfo.

Transition: polyShapeMassInfo stack-allocates a new ShapeMassInfo structure using the inputted values.

Output: polyShapeMassInfo returns the allocated ShapeMassInfo structure.

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 $\vee$ SpaceLocked
spaceRemoveBody	Space*, Body*	-	MainStaticBody $\vee$ BodyNotFound $\vee$ SpaceLocked
spaceContainsShape	Space*, Shape*	Boolean	-
spaceContainsBody	Space*, Body*	Boolean	-
spaceEachBody	Space*, SpaceBodyIteratorFunc, void*	-	-
spaceEachShape	Space*, SpaceShapeIteratorFunc, void*	-	-
spaceReindexStatic	Space*	-	SpaceLocked
spaceReindexShape	Space*, Shape*	-	SpaceLocked
spaceReindexShapesForBody	Space*, Body*	-	SpaceLocked
spacePushFreshContactBuffer	Space*	-	-
contactBufferGetArray	Space*	Contact	-
spacePushContacts	Space*, int	-	BufferOverflow
queryReject	Shape*, Shape*	Boolean	-
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:** spaceAlloc does not accept any inputs.

Exceptions: There are no potential exceptions for spaceAlloc.

Transition: spaceAlloc heap-allocates a new Space object.

Output: spaceAlloc returns a pointer to the allocated Space.

spaceInit: **Input:** spaceInit accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spaceInit.

Transition: spaceInit initializes the inputted Space, allocating new data structures accordingly and zero-initializing all other variables.

Output: spaceInit returns a pointer to the initialized Space.

spaceNew: **Input:** spaceNew does not accept any inputs.

Exceptions: There are no potential exceptions for spaceNew.

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: There are no potential exceptions for spaceDestroy.

Transition: spaceDestroy frees all dynamically-allocated data structures in the inputted Space and their contents, if necessary.

Output: spaceDestroy does not return any value.

spaceFree: **Input:** spaceFree accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spaceFree.

Transition: spaceFree frees the inputted Space and all of its dynamically-allocated variables.

Output: spaceFree does not return any value.

spaceGet: **Input:** Each spaceGet function accepts a Space pointer as input.

Exceptions: There are no potential exceptions for each spaceGet function.

Transition: Each spaceGet function does not make any transition.

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 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 inputted parameter. For spaceSetStaticBody, it also changes the Body's associated Space to the inputted Space.

Output: Each spaceSet function does not return any value.

spaceIsLocked: **Input:** spaceIsLocked accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spaceIsLocked.

Transition: spaceIsLocked makes no transitions.

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 exceptions if the object being added already exists in the inputted Space, if it is already attached to another Space, or if the inputted Space is locked.

Transition: Functions that add CollisionHandlers will initialize new handlers depending on the function and inputted CollisionTypes (or use the default), hash the handler, and add it to the inputted Space's collision handlers. spaceAddShape and spaceAddBody will add the body to the appropriate spatial index and array of the inputted Space, respectively, and the former will update the inputted 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: There are no potential exceptions for spaceFilterArbiters.

Transition: spaceFilterArbiters will remove Arbiters that are associated with the inputted Body and/or Shape from the Space's cached Arbiters.

Output: spaceFilterArbiters does not return any value.

spaceRemove:

Input: Each spaceRemove function accepts a Space pointer and a pointer to the corresponding object as input.

Exceptions: Each spaceRemove function may throw an exception if the object to be removed does not exist within the inputted Space or the Space is locked. Additionally, spaceRemoveBody will throw an exception if the user attempts to remove the Space's designated static Body.

Transition: Each spaceRemove function will detach the object and the inputted Space from each other. Additionally, spaceRemoveShape will detach the Shape from its Body.

Output: Each spaceRemove function does not return any value.

spaceContains:

Input: Each spaceContains function accepts a Space pointer and a pointer to the corresponding object as input.

Exceptions: There are no potential exceptions for each spaceContains function.

Transition: Each spaceContains function makes no transition.

Output: Each spaceContains function returns true if the inputted Space contains the inputted 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: There are no potential exceptions for each spaceEach function.

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 inputted void pointer.

Output: Each spaceEach function does not return any value.

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 an exception if the inputted Space is locked.

Transition: spaceReindexStatic reindexes all the static Shapes for the Space. spaceReindexShape will only reindex the inputted Shape in its spatial index, and spaceReindexShapesForBody will reindex all the shapes attached to the inputted Body in their respective spatial indices.

Output: Each spaceReindex function does not return any value.

spacePushFresh Contact Buffer: **Input:** spacePushFreshContactBuffer accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spacePushFreshContactBuffer.

Transition: spacePushFreshContactBuffer initializes a new ContactBufferHeader and set it to the inputted Space's contact buffer head.

Output: spacePushFreshContactBuffer does not return any value.

contactBufferGetArray:

Input: contactBufferGetArray accepts a Space pointer as input.

Exceptions: There are no potential exceptions for contactBufferGetArray.

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.

spacePushContacts:

Input: spacePushContacts accepts a Space pointer and an integer as input.

Exceptions: spacePushContacts may throw an exception if the inputted integer exceeds the MAX_CONTACTS_PER_ARBITER.

Transition: spacePushContacts increases the number of Contacts of the inputted Space's contact buffer head by the given integer.

Output: spacePushContacts does not return any value.

queryReject:

Input: queryReject accepts two Shape pointers as input.

Exceptions: There are no potential exceptions for queryReject.

Transition: queryReject tests for collision conditions. In particular, it tests if the bounding boxes of both inputted Shapes overlap and if they belong to different Bodies.

Output: queryReject returns true if any of the above tests fail, and false otherwise.

spaceCollideShapes:

Input: spaceCollideShapes accepts two Shape pointers, a Collision ID and a Space pointer as inputs.

Exceptions: There are no potential exceptions for spaceCollideShapes.

Transition: spaceCollideShapes tests if the inputted Shapes can be collided using queryReject. If it fails, it returns the inputted 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 inputted Space, updates the Arbiter for the inputted 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: There are no potential exceptions for spaceArbiterSetFilter.

Transition: spaceArbiterSetFilter caches the inputted 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: There are no potential exceptions for spaceLock.

Transition: spaceLock locks the inputted Space by modifying its locked variable.

Output: spaceLock does not return any value.

spaceUnlock:

Input: spaceUnlock accepts a Space pointer and a Boolean value as input.

Exceptions: spaceUnlock may throw an exception if the locked field of the inputted Space falls to a negative value during the function's transition.

Transition: spaceUnlock will unlock the inputted Space by modifying its locked variable. If the inputted Boolean is true, and the Space is set to not skip post-step callbacks, the function will also run those callbacks.

Output: spaceUnlock does not return any value.

spaceArrayForBodyType:

Input: spaceArrayForBodyType accepts a Space pointer and a BodyType value as inputs.

Exceptions: There are no potential exceptions for spaceArrayForBodyType.

Transition: spaceArrayForBodyType makes no transitions.

Output: spaceArrayForBodyType returns an Array pointer to the inputted Space's array of bodies, corresponding to the inputted BodyType.

shapeUpdateFunc:

Input: shapeUpdateFunc accepts a Shape and a void pointer as inputs.

Exceptions: There are no potential exceptions for spaceUpdateFunc.

Transition: shapeUpdateFunc calls updates and caches the inputted Shape's BB.

Output: shapeUpdateFunc does not return any value.

spaceStep:

Input: spaceStep accepts a Space pointer and a double as inputs.

Exceptions: There are no potential exceptions for spaceStep.

Transition: spaceStep updates the inputted Space following the specified timestep (inputted 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: spaceStep does not return any value.

6.4.4 Local Constants

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

6.4.5 Local Functions

arbiterSetEql:

Input: arbiterSetEql accepts a pointer to an array of Shape pointers and an Arbiter pointer as inputs.

Exceptions: There are no potential exceptions for arbiterSetEql.

Transition: arbiterSetEql makes no transition.

Output: arbiterSetEql returns true if the Shapes in the inputted array are equal to the inputted Arbiter's Shapes, and false otherwise.

handlerSetEql:

Input: handlerSetEql accepts two CollisionHandler pointers as inputs.

Exceptions: There are no potential exceptions for handlerSetEql.

Transition: handlerSetEql makes no transition.

Output: handlerSetEql returns true if the CollisionTypes of both handlers are equal, and false otherwise.

handlerSet Trans:

Input: handlerSetTrans accepts a CollisionHandler pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for handlerSetTrans.

Transition: handlerSetTrans clones the inputted CollisionHandler.

Output: handlerSetTrans returns a void pointer to the cloned handler.

default:

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

Exceptions: There are no potential exceptions for each default function.

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 inputted Arbiter and Space, defaultPreSolve calls arbiterCallWildcardPreSolveA and arbiterCallWildcardPreSolveB, and so on.

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: There are no potential exceptions for alwaysCollide.

Transition: alwaysCollide makes no transition.

Output: alwaysCollide always returns true.

doNothing: **Input:** doNothing takes an Arbiter pointer, a Space pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for doNothing.

Transition: doNothing makes no transition.

Output: doNothing does not return any value.

shapeVelocity Func: **Input:** shapeVelocityFunc accepts a Shape pointer as input.

Exceptions: There are no potential exceptions for shapeVelocityFunc.

Transition: shapeVelocityFunc makes no transition.

Output: shapeVelocityFunc returns the velocity function of the Body associated to the inputted Shape.

freeWrap: **Input:** freeWrap accepts two void pointers as inputs.

Exceptions: There are no potential exceptions for freeWrap.

Transition: freeWrap frees the first inputted pointer.

Output: freeWrap does not return any value.

spaceUseWildcardDefaultHandler:

Input: spaceUseWildcardDefaultHandler accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spaceUseWildcardDefaultHandler.

Transition: The function sets the Space to use wildcards and copies collisionHandlerDefault to the Space's default handler.

Output: spaceUseWildcardDefaultHandler does not return any value.

cachedArbitersFilter:

Input: cachedArbitersFilter accepts an Arbiter pointer and a pointer to an ArbiterFilterContext structure.

Exceptions: There are no potential exceptions for cachedArbitersFilter.

Transition: cachedArbitersFilter is the filtering function used by spaceFilterArbiters to remove and recycle cached Arbiters that are associated with the Body and/or Shape defined in the inputted ArbiterFilterContext structure.

Output: cachedArbitersFilter returns true if no Arbiters were removed, and false otherwise.

spaceEachShapeIterator:

Input: spaceEachShapeIterator accepts a Shape pointer and a pointer to a SpaceShapeContext structure as inputs.

Exceptions: There are no potential exceptions for spaceEachShapeIterator.

Transition: spaceEachShapeIterator calls the function in the inputted SpaceShapeContext structure with the inputted Shape and the data pointer defined in the structure.

Output: spaceEachShapeIterator does not return any value.

spaceAllocContactBuffer:

Input: spaceAllocContactBuffer accepts a Space pointer as input.

Exceptions: There are no potential exceptions for spaceAllocContactBuffer.

Transition: spaceAllocContactBuffer heap-allocates a new contact buffer and adds it to the inputted Space's allocated buffers

Output: spaceAllocContactBuffer returns a pointer to the allocated ContactBuffer as output, cast as a ContactBufferHeader pointer.

contactBufferHeaderInit:

Input: contactBufferHeaderInit accepts a ContactBufferHeader pointer, a Timestamp and another ContactBufferHeader pointer as input.

Exceptions: There are no potential exceptions for contactBufferHeaderInit.

Transition: contactBufferHeaderInit initializes the first inputted ContactBufferHeader. It modifies its timestamp to the given Timestamp, its next header to be the next header of the second inputted ContactBufferHeader (or to the first inputted header if the second one is null), and its number of Contacts to zero.

Output: contactBufferHeaderInit returns a pointer to the initialized ContactBufferHeader.

spacePopContacts:

Input: spacePopContacts accepts a Space pointer and an integer as inputs.

Exceptions: There are no potential exceptions for spacePopContacts.

Transition: spacePopContacts decrements the number of Contacts of the inputted Space's contact buffer head by the inputted integer.

Output: spacePopContacts does not return any value.

spaceArbiterSetTrans:

Input: spaceArbiterSetTrans accepts a pointer to an array of Shape pointers and a Space pointer.

Exceptions: spaceArbiterSetTrans may throw an InsufficientBufferSize exception if the inputted Space has no pooled Arbiters and the size of an Arbiter object exceeds the buffer size (BUFFER_BYTES).

Transition: If the inputted 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 inputted array.

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	ContactIndexOutOfBounds
arbiterGetPointB	Arbiter*, int	Vector	ContactIndexOutOfBounds
arbiterGetDepth	Arbiter*, int	double	ContactIndexOutOfBounds
arbiterGetContactPointSet	Arbiter*	ContactPointSet	-
arbiterGetShapes	Arbiter*, Shape**, Shape**	-	
arbiterGetBodies	Arbiter*, Body**, Body**	-	
arbiterSetRestitution	Arbiter*, double	-	-
arbiterSetFriction	Arbiter*, double	-	-
arbiterSetSurfaceVelocity	Arbiter*, Vector	-	-
arbiterSetContactPointSet	Arbiter*, ContactPointSet*	-	ImmutableNumContacts
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 ∈ {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: $\mathbb{R}$
fric: $\mathbb{R}$
surfaceVel: Vector
a: Shape* (constant)
b: Shape* (constant)
bodyA: Body*
bodyB: Body*
threadA: ArbiterThread
threadB: ArbiterThread
count: $\mathbb{Z}$
contacts: Contact*
normal: Vector
handler: CollisionHandler*
handlerA: CollisionHandler*
handlerB: CollisionHandler*
swapped: $\mathbb{B}$
stamp: Timestamp
state: ArbiterState

ContactPointSet:

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

7.4.2 Assumptions

All inputted 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: There are no potential exceptions for arbiterInit.

Transition: arbiterInit initializes the inputted Arbiter. Its state is set to ARBITER_STATE_FIRST_COLLISION. Its Shapes and Bodies are set to the inputted Shapes and their associated Bodies, and all other fields are zero-initialized.

Output: arbiterInit returns a pointer to the initialized Arbiter as output.

arbiterThreadForBody:

Input: arbiterThreadForBody accepts an Arbiter pointer and a Body pointer as inputs.

Exceptions: There are no potential exceptions for arbiterThreadForBody.

Transition: arbiterThreadForBody makes no transition.

Output: arbiterThreadForBody returns the inputted Arbiter's ArbiterThread which corresponds to the inputted Body.

arbiterUnthread:

Input: arbiterUnthread accepts an Arbiter pointer as input.

Exceptions: There are no potential exceptions for arbiterUnthread.

Transition: arbiterUnthread calls unthreadHelper on the inputted Arbiter's Bodies to remove this Arbiter from the thread.

Output: arbiterUnthread does not return any value.

arbiterUpdate:

Input: arbiterUpdate accepts an Arbiter pointer, a CollisionInfo pointer and a Space pointer as inputs.

Exceptions: There are no potential exceptions for arbiterUpdate.

Transition: arbiterUpdate updates the Arbiter's state after a collision using the inputted CollisionInfo and Space. If the Arbiter had been cached, it changes the state to ARBITER_STATE_FIRST_COLLISION.

Output: arbiterUpdate does not return any value.

arbiterPreStep:

Input: arbiterPreStep accepts an Arbiter pointer and three doubles as inputs.

Exceptions: There are no potential exceptions for arbiterPreStep.

Transition: arbiterPreStep calculates the mass normal, mass tangent, bias velocity and bounce velocity for each of the inputted Arbiter's contacts, using the three inputted doubles which represent the timestep, collision slop and collision bias, respectively.

Output: arbiterPreStep does not return any value.

arbiterApply CachedImpulse:

Input: arbiterApplyCachedImpulse accepts an Arbiter pointer and a double as inputs.

Exceptions: There are no potential exceptions for arbiterApplyCachedImpulse.

Transition: arbiterApplyCachedImpulse applies the impulses stored in the inputted Arbiter's contacts to its Bodies, using the inputted double as a timestep coefficient.

Output: arbiterApplyCachedImpulse does not return any value.

arbiterApply Impulse:

Input: arbiterApplyImpulse accepts an Arbiter pointer as input.

Exceptions: There are no potential exceptions for arbiterApplyImpulse.

Transition: arbiterApplyImpulse applies all impulses stored in the inputted Arbiter's contacts to its Bodies.

Output: arbiterApplyImpulse does not return any value.

arbiterNext:

Input: arbiterNext accepts an Arbiter pointer and a Body pointer as inputs.

Exceptions: There are no potential exceptions for arbiterNext.

Transition: arbiterNext makes no transition.

Output: arbiterNext returns a pointer to the next Arbiter from the inputted Arbiter's ArbiterThread which corresponds to the inputted 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 an exception when the inputted integer exceeds the number of contact points for the inputted Arbiter.

Transition: arbiterGetContactPointSet initializes a new ContactPointSet for the inputted Arbiter using its array of Contacts. arbiterGetShapes and arbiterGetBodies retrieve the inputted Arbiter's Shapes and Bodies, respectively, and store them in the inputted 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 exception if the number of contact points in the inputted ContactPointSet differs from the current number of contact points of the inputted Arbiter.

Transition: Each arbiterSet function sets the value of their corresponding parameter with the inputted value. In particular, arbiterSetContactPointSet modifies the contents of the inputted Arbiter's array of Contacts according to the inputted ContactPointSet.

Output: Each arbiterSet function does not return any value.

arbiterIs:

Input: Each arbiterIs function accepts an Arbiter pointer as input.

Exceptions: There are no potential exceptions for each arbiterIs function.

Transition: Each arbiterIs function makes no transition.

Output: Each arbiterIs function checks the state of the inputted Arbiter and returns a Boolean value according to the result.

arbiterIgnore:

Input: arbiterIgnore accepts an Arbiter pointer as input.

Exceptions: There are no potential exceptions for arbiterIgnore.

Transition: arbiterIgnore sets the state of the inputted Arbiter to ARBITER_STATE_IGNORE.

Output: arbiterIgnore always returns false.

arbiterTotal:

Input: each arbiterTotal function accepts an Arbiter pointer as input.

Exceptions: There are no potential exceptions for arbiterTotal.

Transition: Each arbiterTotal function iterates through the inputted Arbiter's array of Contacts to compute the total quantity of the corresponding parameter (impulse or kinetic energy) contained in the Arbiter.

Output: Each arbiterTotal function returns the value of their corresponding parameter.

arbiterCall Wildcard:

Input: Each arbiterCallWildcard function accepts an Arbiter pointer and a Space pointer as inputs.

Exceptions: There are no potential exceptions for each arbiterCallWildcard function.

Transition: Each arbiterCallWildcard function calls the corresponding function from the inputted Arbiter's CollisionHandlers. The input arguments will be the inputted Arbiter, the inputted 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: There are no potential exceptions for unthreadHelper.

Transition: unthreadHelper removes the inputted Arbiter from the ArbiterThread corresponding to the inputted Body. It may also remove the Arbiter from the Body.

Output: unthreadHelper does not return any value.

spaceLookup Handler:

Input: spaceLookupHandler accepts a Space pointer, two CollisionType values and a CollisionHandler pointer as inputs.

Exceptions: There are no potential exceptions for spaceLookupHandler.

Transition: spaceLookupHandler searches the inputted Space's collision handlers for a handler corresponding to the inputted CollisionTypes.

Output: spaceLookupHandler returns a pointer to the retrieved CollisionHandler. If not found, it returns the inputted CollisionHandler pointer.

8 MIS of the Control Module

8.1 Module Name: Chipmunk

8.2 Uses

This module only uses C's 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 $\quad 1 \times 10^{-37}$

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

Due to the way strings are represented in C, all strings in the following table are equivalent to a pointer to a constant character array (const char*).

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: There are no potential exceptions for message.

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

Output: message does not return any value.

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: There are no potential exceptions for each assert macro; they are used to throw exceptions.

Transition: Each assert macro tests the inputted Boolean expression. If the test fails, each macro will print the inputted 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 inputted error value instead of aborting the program.

Output: Each assert macro does not return any value normally. In UNIT_TEST mode, they may return the inputted error value.

fclamp:

Input: Each fclamp function accepts a double as input. The regular fclamp accepts two additional doubles as inputs.

Exceptions: There are no potential exceptions for each fclamp function.

Transition: Each fclamp function makes no transition.

Output: fclamp 'clamps' the first inputted double between the second and third inputted 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 clamps the inputted double between 0 and 1. Each fclamp function returns a double.

flerp:

Input: Each flerp function accepts three doubles as inputs.

Exceptions: There are no potential exceptions for each flerp function.

Transition: Each flerp function makes no transition.

Output: flerp linearly interpolates between the first two inputted doubles for a percentage specified by the third inputted double. Similarly, flerpconst linearly interpolates between the first two doubles by no more than a constant specified in the third inputted 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: There are no potential exceptions for loopIndices.

Transition: loopIndices iterates through the points contained in the inputted Vector array; the length of the array should be equal to the inputted 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 inputted integer pointers, respectively.

Output: loopIndices does not return any value.

convexHull:

Input: convexHull accepts an integer, two pointers to Vector arrays, an integer pointer and a double as inputs.

Exceptions: There are no potential exceptions for convexHull.

Transition: convexHull calculates the hull size given by the set of points contained in the first inputted 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: There are no potential exceptions for QHullPartition.

Transition: QHullPartition partitions the set of points in the convex hull for reduction.

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: There are no potential exceptions for QHullReduce.

Transition: QHullReduce simplifies (shrinks) the convex hull by a certain tolerance, which is specified by the inputted double.

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 C's 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: There are no potential exceptions for vect.
Transition: vect allocates a new Vector using the inputted doubles.
Output: vect returns the allocated Vector as output.

vectEqual:
Input: vectEqual accepts two Vectors as input.
Exceptions: There are no potential exceptions for vectEqual.
Transition: vectEqual compares the values of the inputted Vectors.
Output: vectEqual returns the Boolean result of the above test.

vectAdd:
Input: vectAdd accepts two Vectors as input.
Exceptions: There are no potential exceptions for vectAdd.
Transition: vectAdd calculates the sum of the inputted Vectors.
Output: vectAdd returns the result as a Vector.

vectSub:
Input: vectSub accepts two Vectors as input.
Exceptions: There are no potential exceptions for vectSub.
Transition: vectSub calculates the difference of the inputted Vectors.
Output: vectSub returns the result as a Vector.

vectMult:
Input: vectMult accepts a Vector and a double as inputs.
Exceptions: There are no potential exceptions for vectMult.
Transition: vectMult calculates the scalar multiple of the inputted Vector with the inputted double.
Output: vectMult returns the result as a Vector.

vectNeg:

Input: vectNeg accepts a Vector as input.

Exceptions: There are no potential exceptions for vectNeg.

Transition: vectNeg negates the inputted Vector.

Output: vectNeg returns the result as a Vector.

vectDot:

Input: vectDot accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for vectDot.

Transition: vectDot calculates the dot product of the inputted Vectors.

Output: vectDot returns the result as a double.

vectCross:

Input: vectCross accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for vectCross.

Transition: vectCross calculates the cross product of the inputted Vectors.

Output: vectCross returns the z-component of the product as a double.

vectPerp:

Input: vectPerp accepts a Vector as input.

Exceptions: There are no potential exceptions for vectPerp.

Transition: vectPerp rotates the inputted Vector by 90 degrees clockwise.

Output: vectPerp returns the rotated Vector as output.

vectRPerp:

Input: vectRPerp accepts a Vector as input.

Exceptions: There are no potential exceptions for vectRPerp.

Transition: vectRPerp rotates the inputted Vector by 90 degrees anti-clockwise.

Output: vectRPerp returns the rotated vector as output.

vectProject:

Input: vectProject accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for vectProject.

Transition: vectProject projects the first inputted Vector onto the second.

Output: vectProject returns the projected Vector as output.

vectForAngle:

Input: vectForAngle accepts a double as input.

Exceptions: There are no potential exceptions for vectForAngle.

Transition: vectForAngle computes the Vector corresponding to the inputted angle (double), measured from the x-axis.

Output: vectForAngle returns the result as a Vector.

vectToAngle:

Input: vectToAngle accepts a Vector as input.

Exceptions: There are no potential exceptions for vectToAngle.

Transition: vectToAngle calculates the angle between the inputted Vector and the x-axis.

Output: vectToAngle returns the result as a double.

vectRotate:

Input: vectRotate accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for vectRotate.

Transition: vectRotate rotates the first inputted Vector by the second using complex multiplication.

Output: vectRotate returns the new rotated Vector as output.

vectUnrotate:

Input: vectUnrotate accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for vectUnrotate.

Transition: vectUnrotate is the inverse of vectRotate.

Output: vectUnrotate returns the original Vector before it was rotated by another Vector using vectRotate.

vectLength:

Input: Each vectLength function accepts a Vector as input.

Exceptions: There are no potential exceptions for each vectLength function.

Transition: vectLength and vectLengthSq calculates the regular and squared length of the inputted Vector, respectively.

Output: Each vectLength function returns the result as a double.

vectNormalize:

Input: vectNormalize accepts a Vector as input.

Exceptions: There are no potential exceptions for vectNormalize.

Transition: vectNormalize converts the inputted Vector into a unit vector.

Output: vectNormalize returns the normalized Vector as output.

vectClamp:

Input: vectClamp accepts a Vector and a double as inputs.

Exceptions: There are no potential exceptions for vectClamp.

Transition: vectClamp clamps the inputted Vector to a length specified by the inputted double. If the length of the inputted Vector is less than the inputted length, vectClamp returns the inputted Vector. Otherwise, it shrinks the Vector to the specified length.

Output: vectClamp returns the result as a Vector.

vectLerp:

Input: vectLerp accepts two Vectors and a double as inputs.

Exceptions: There are no potential exceptions for vectLerp.

Transition: vectLerp linearly interpolates between the two inputted Vectors for a percentage specified by the inputted double.

Output: vectLerp returns the new interpolated Vector as output.

vectDist:

Input: Each vectDist function accepts two Vectors as input.

Exceptions: There are no potential exceptions for each vectDist function.

Transition: vectDist and vectDistSq calculates the regular and squared distance between the two inputted Vectors.

Output: Each vectDist function returns the result as a double.

vectNear:

Input: vectNear accepts two Vectors and a double as input.

Exceptions: There are no potential exceptions for vectNear.

Transition: vectNear checks if the distance between the inputted Vectors is less than the distance specified by the inputted double.

Output: vectNear returns the Boolean result of the above test.

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: ℝ
right: ℝ
bottom: ℝ
top: ℝ

10.4.2 Access Program Semantics

BBNew:

Input: BBNew accepts four doubles as input.

Exceptions: There are no potential exceptions for BBNew.

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

Output: BBNew returns the allocated BB as output.

BBNewFor Extents:

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

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

Transition: BBNewForExtents creates a new BB centered on the inputted Vector. The BB's dimensions are calculated from the inputted doubles, which provide the half-width and half-height, respectively.

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 inputted double is negative.

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

Output: BBNewForCircle returns the new BB as output.

BBIntersects:

Input: BBIntersects accepts two BBs as inputs.

Exceptions: There are no potential exceptions for BBIntersects.

Transition: BBIntersects checks if the inputted BBs intersect one another.

Output: BBIntersects returns the Boolean result of the above test.

BBContainsBB: **Input:** BBContainsBB accepts two BBs as inputs.

Exceptions: There are no potential exceptions for BBContainsBB.

Transition: BBContainsBB checks if the first inputted BB contains the second.

Output: BBContainsBB returns the Boolean result of the above test.

BBContains Vect: **Input:** BBContainsVect accepts a BB and a Vector as inputs.

Exceptions: There are no potential exceptions for BBContainsVect.

Transition: BBContainsVect checks if the inputted Vector is contained in the inputted BB.

Output: BBContainsVect returns the Boolean result of the above test.

BBMerge: **Input:** BBMerge accepts two BBs as inputs.

Exceptions: There are no potential exceptions for BBMerge.

Transition: BBMerge creates a new BB containing the two inputted BBs.

Output: BBMerge returns the new BB as output.

BBCenter: **Input:** BBCenter accepts a BB as input.

Exceptions: There are no potential exceptions for BBCenter.

Transition: BBCenter computes the centroid of the inputted BB.

Output: BBCenter returns the result as a Vector.

BBArea: **Input:** BBArea accepts a BB as input.

Exceptions: There are no potential exceptions for BBArea.

Transition: BBArea calculates the area of the inputted BB.

Output: BBArea returns the result as a double.

BBMergedArea: **Input:** BBMergedArea accepts two BBs as inputs.

Exceptions: There are no potential exceptions for BBMergedArea.

Transition: BBMergedArea calculates the area of the region containing both inputted BBs.

Output: BBMergedArea returns the result as a double.

BBClampVect: **Input:** BBClampVect accepts a BB and a Vector as inputs.

Exceptions: There are no potential exceptions for BBClampVect.

Transition: BBClampVect clamps the inputted Vector to the dimensions of the inputted BB.

Output: BBClampVect returns the clamped Vector as output.

BBWrapVect: **Input:** BBWrapVect accepts a BB and a Vector as inputs.

Exceptions: There are no potential exceptions for BBWrapVect.

Transition: BBWrapVect wraps the inputted Vector to the inputted BB.

Output: BBWrapVect returns the wrapped Vector as output.

BBOffset: **Input:** BBOffset accepts a BB and a Vector as inputs.

Exceptions: There are no potential exceptions for BBOffset.

Transition: BBOffset translates the inputted BB by the specified Vector.

Output: BBOffset 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: There are no potential exceptions for each transformNew function.

Transition: Each transformNew function creates a new Transform matrix from the inputted doubles.

Output: transformNew and transformNewTranspose returns the new Transform matrix in regular and transposed order, respectively.

transformInverse:

Input: transformInverse accepts a Transform matrix as input.

Exceptions: There are no potential exceptions for transformInverse.

Transition: transformInverse calculates the inverse of the inputted Transform matrix.

Output: transformInverse returns the result as a Transform matrix.

transformMult:

Input: transformMult accepts two Transform matrices as inputs.

Exceptions: There are no potential exceptions for transformMult.

Transition: transformMult multiplies the two inputted Transform matrices together.

Output: transformMult returns the result as a Transform matrix.

transformPoint:

Input: transformPoint accepts a Transform matrix and a Vector as inputs.

Exceptions: There are no potential exceptions for transformPoint.

Transition: transformPoint applies the affine transformation from the inputted Transform matrix to the inputted Vector.

Output: transformPoint returns the result as a Vector.

transformVect:

Input: transformVect accepts a Transform matrix and a Vector as inputs.

Exceptions: There are no potential exceptions for transformVect.

Transition: transformVect applies the linear transformation from the inputted Transform matrix to the inputted Vector.

Output: transformVect returns the result as a Vector.

transformBB:

Input: transformBB accepts a Transform matrix and a BB as inputs.

Exceptions: There are no potential exceptions for transformBB.

Transition: transformBB calculates the half-dimensions of the inputted BB, applies the inputted 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 inputted 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: There are no potential exceptions for transformTranslate.

Transition: transformTranslate creates a translation matrix from the inputted Vector.

Output: transformTranslate returns the new Transform matrix as output.

transformScale:

Input: transformScale accepts two doubles as inputs.

Exceptions: There are no potential exceptions for transformScale.

Transition: transformScale creates a scaling matrix from the inputted 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: There are no potential exceptions for transformRotate.

Transition: transformRotate calculates a Vector from the angle specified by the inputted 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: There are no potential exceptions for transformRigid.

Transition: transformRigid calculates a Vector from the angle specified by the inputted double and creates a rigid transformation matrix from the inputted parameters, using the computed Vector for the rotation components and the inputted 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: There are no potential exceptions for transformRigidInverse.

Transition: transformRigidInverse makes no transition.

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*, SpatialIndexClass*, SpatialIndexBBFunc, SpatialIndex*	SpatialIndex*	AttachedStaticIndex
spatialIndexFree	SpatialIndex*	-	-
spatialIndexCollideStatic	SpatialIndex*, SpatialIndex*, SpatialIndexQueryFunc, 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, SpatialIndex-QueryFunc, void*	-	-
spatialIndexReindexQuery	SpatialIndex*, SpatialIndex-QueryFunc, 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
staticIndex: SpatialIndex*
queryFunc: SpatialIndexQueryFunc
data: void*

12.4.2 Assumptions

spatialIndexInit is called before any other access program. All inputted 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 exception if the last inputted SpatialIndex pointer for the staticIndex is already associated to another dynamicIndex.

Transition: spatialIndexInit initializes the first inputted SpatialIndex with the inputted 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: There are no potential exceptions for spatialIndexFree.

Transition: spatialIndexFree frees the inputted SpatialIndex.

Output: spatialIndexFree does not return any value.

spatialIndex CollideStatic:

Input: spatialIndexCollideStatic accepts two SpatialIndex pointers, a SpatialIndexQueryFunc function pointer, and a void pointer as inputs.

Exceptions: There are no potential exceptions for spatialIndexCollideStatic.

Transition: If the second inputted SpatialIndex is non-empty, the function creates a new DynamicToStaticContext using the inputted parameters; the bbfunc field is set to the bbfunc of the first inputted SpatialIndex. It will then iterate through the first SpatialIndex using dynamicToStaticIter and the new context.

Output: spatialIndexCollideStatic does not return any value.

spatialIndex Destroy:

Input: spatialIndexDestroy accepts a SpatialIndex pointer as input.

Exceptions: There are no potential exceptions for spatialIndexDestroy.

Transition: spatialIndexDestroy calls the internal destroying function from the class of the inputted SpatialIndex with the index itself.

Output: spatialIndexDestroy does not return any value.

spatialIndex Count:

Input: spatialIndexCount accepts a SpatialIndex pointer as input.

Exceptions: There are no potential exceptions for spatialIndexCount.

Transition: spatialIndexCount calls the internal counting function from the class of the inputted SpatialIndex 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: There are no potential exceptions for spatialIndexEach.

Transition: spatialIndexEach calls the internal iterator function from the class of the inputted SpatialIndex with the inputted parameters.

Output: spatialIndexEach does not return any value.

spatialIndex Contains:

Input: spatialIndexContains accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for spatialIndexContains.

Transition: spatialIndexContains calls the internal presence-checking function from the class of the inputted SpatialIndex with the inputted parameters.

Output: spatialIndexContains returns true if the inputted index contains the object in the inputted void pointer, and false otherwise.

spatialIndex Insert:

Input: spatialIndexInsert accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for spatialIndexInsert.

Transition: spatialIndexInsert calls the internal insertion function from the inputted SpatialIndex's class with the inputted parameters.

Output: spatialIndexInsert does not return any value.

spatialIndex Remove:

Input: spatialIndexRemove accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for spatialIndexRemove.

Transition: spatialIndexRemove calls the internal removal function from the inputted SpatialIndex's class with the inputted parameters.

Output: spatialIndexRemove does not return any value.

spatialIndex Reindex:

Input: spatialIndexReindex accepts a SpatialIndex pointer as input.

Exceptions: There are no potential exceptions for spatialIndexReindex.

Transition: spatialIndexReindex calls the internal reindexing function from the inputted SpatialIndex's class with the index itself.

Output: spatialIndexReindex does not return any value.

spatialIndex ReindexObject:

Input: spatialIndexReindexObject accepts a SpatialIndex pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for spatialIndexReindexObject.

Transition: spatialIndexReindexObject calls the internal object-reindexing function from the inputted SpatialIndex's class with the inputted parameters.

Output: spatialIndexReindexObject does not return any value.

spatialIndex Query:

Input: spatialIndexQuery accepts a SpatialIndex pointer, a void pointer, a BB and a SpatialIndexQueryFunc function pointer.

Exceptions: There are no potential exceptions for spatialIndexQuery.

Transition: spatialIndexQuery calls the internal querying function from the inputted SpatialIndex's class with the inputted parameters.

Output: spatialIndexQuery does not return any value.

spatialIndexReindexQuery:

Input: spatialIndexReindexQuery accepts a SpatialIndex pointer, a SpatialIndexQueryFunc function pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for spatialIndexReindexQuery.

Transition: spatialIndexReindexQuery calls the internal query-based reindexing function from the inputted SpatialIndex's class with the inputted parameters.

Output: spatialIndexReindexQuery does not return any value.

12.4.4 Local Functions

dynamicToStaticIter:

Input: dynamicToStaticIter accepts a void pointer and a DynamicToStaticContext pointer as inputs.

Exceptions: There are no potential exceptions for dynamicToStaticIter.

Transition: dynamicToStaticIter queries the index by calling spatialIndexQuery with the inputted void pointer and the fields of the inputted DynamicToStaticContext.

Output: dynamicToStaticIter does not return any value.

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* × Shape* × CollisionInfo* → void
SupportPointFunc : Shape* × Vector → 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
b: Vector
n: Vector
d: $\mathbb{R}$
id: CollisionID

13.4.2 Access Program Semantics

relative_velocity:

Input: relative_velocity accepts two Body pointers and two Vectors as inputs.

Exceptions: There are no potential exceptions for relative_velocity.

Transition: relative_velocity calculates the relative velocity of the second inputted Body relative to the first inputted Body with the inputted parameters.

Output: relative_velocity returns the result as a Vector.

normal_relative_velocity:

Input: normal_relative_velocity accepts two Body pointers and three Vectors as inputs.

Exceptions: There are no potential exceptions for normal_relative_velocity.

Transition: normal_relative_velocity calculates the dot product of the relative velocity between the two inputted Bodies and the normal (third inputted Vector).

Output: normal_relative_velocity returns the result as a double.

apply_impulse:

Input: apply_impulse accepts a Body pointer and two Vectors as inputs.

Exceptions: There are no potential exceptions for apply_impulse.

Transition: apply_impulse recalculates the inputted Body's linear and angular velocity using the impulse (first inputted Vector) and point of application (second inputted Vector).

Output: apply_impulse does not return any value.

apply_impulses:

Input: apply_impulses accepts two Body pointers and three Vectors as inputs.

Exceptions: There are no potential exceptions for apply_impulses.

Transition: apply_impulses applies the inputted impulse (third inputted Vector) to the two inputted Bodies, in opposite directions, to recalculate their linear and angular velocities, using their points of application (first and second inputted Vectors).

Output: apply_impulses does not return any value.

apply_bias_impulse:

Input: apply_bias_impulse accepts a Body pointer and two Vectors as inputs.

Exceptions: There are no potential exceptions for apply_bias_impulse.

Transition: apply_bias_impulse recalculates the inputted Body's linear and angular bias velocities using the impulse (first inputted Vector) and point of application (second inputted Vector).

Output: apply_bias_impulse does not return any value.

apply_bias_impulses:

Input: apply_bias_impulses accepts two Body pointers and three Vectors as inputs.

Exceptions: There are no potential exceptions for apply_bias_impulses.

Transition: apply_bias_impulses applies the inputted impulse (third inputted Vector) to the two inputted Bodies, in opposite directions, to recalculate their linear and angular bias velocities, using their points of application (first and second inputted Vectors).

Output: apply_bias_impulses does not return any value.

k_scalar_body:

Input: k_scalar_body accepts a Body pointer and two Vectors as inputs.

Exceptions: There are no potential exceptions for k_scalar_body.

Transition: k_scalar_body first calculates the cross product of the two inputted Vectors. Then, it computes the product of the inverse momentum of the inputted Body and the squared cross product of the inputted Vectors. Finally, it calculates the sum of this quantity and the Body's inverse mass.

Output: k_scalar_body 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 exception if the calculated value is equal to zero.

Transition: k_scalar calculates k_scalar_body for the first inputted Body with the first and last inputted Vector, and for the second inputted Body with the second and last inputted Vector. It then calculates the sum of these results.

Output: k_scalar returns the above sum as a double.

collide:

Input: collide accepts two Shape pointers, a CollisionID and a Contact pointer as inputs.

Exceptions: There are no potential exceptions for collide.

Transition: collide creates a new CollisionInfo structure with the inputted 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: There are no potential exceptions for shapesCollide.

Transition: shapesCollide declares a new Contact array and generates a CollisionInfo structure for the inputted 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

checkArea:

Input: checkArea accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for checkArea.

Transition: checkArea calculates the product of the first Vector's x-component with the second's y-component, and the product of the first Vector's y-component with the second's x-component.

Output: checkArea returns true if the first result is greater than the second, and false otherwise.

checkSignedArea:

Input: checkSignedArea accepts three Vectors as inputs.

Exceptions: There are no potential exceptions for checkSignedArea.

Transition: checkSignedArea calculates the signed area of a triangle using the three inputted Vectors as its vertices.

Output: checkSignedArea returns true if the signed area is positive, and false otherwise.

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 inputted CollisionInfo exceeds MAX_CONTACTS_PER_ARBITER.

Transition: collisionInfoPushContact pushes a new Contact structure into the inputted CollisionInfo's Contacts array with the other inputted parameters and updates its number of Contacts accordingly.

Output: collisionInfoPushContact does not return any value.

polySupport PointIndex:

Input: polySupportPointIndex accepts an integer, a pointer to a SplittingPlane array and a Vector as inputs.

Exceptions: There are no potential exceptions for polySupportPointIndex.

Transition: polySupportPointIndex iterates through the inputted array. For each point in the array, the function calculates the dot product of the point's Vector with the inputted Vector, and computes the index of the point that maximizes this quantity.

Output: polySupportPointIndex returns the index as an integer.

supportPoint New:

Input: supportPointNew accepts a Vector and a CollisionID as inputs.

Exceptions: There are no potential exceptions for supportPointNew.

Transition: supportPointNew allocates a new SupportPoint structure with the inputted parameters.

Output: supportPointNew returns the allocated SupportPoint structure as output.

circleSupport Point:

Input: circleSupportPoint accepts a CircleShape pointer and a Vector as inputs.

Exceptions: There are no potential exceptions for circleSupportPoint.

Transition: circleSupportPoint creates a new SupportPoint with the inputted CircleShape's transformed center and zero as the ID.

Output: circleSupportPoint returns the new SupportPoint as output.

segmentSupport Point:

Input: segmentSupportPoint accepts a SegmentShape pointer and a Vector as inputs.

Exceptions: There are no potential exceptions for segmentSupportPoint.

Transition: segmentSupportPoint calculates the dot product of the inputted SegmentShape's endpoints with the inputted Vector. It creates a new SupportPoint with the endpoint that maximizes the product and either zero or one as the ID, depending on the endpoint used.

Output: segmentSupportPoint returns the new SupportPoint as output.

polySupport Point:

Input: polySupportPoint accepts a PolyShape pointer and a Vector as input.

Exceptions: There are no potential exceptions for polySupportPoint.

Transition: polySupportPoint finds the index of the inputted PolyShape's support point with polySupportPointIndex and creates a new SupportPoint, using the vertex corresponding to the index and the index itself as the ID.

Output: polySupportPoint returns the new SupportPoint as output.

minkowskiPointNew:

Input: minkowskiPointNew accepts two SupportPoints as input.

Exceptions: There are no potential exceptions for minkowskiPointNew.

Transition: minkowskiPointNew allocates a new MinkowskiPoint structure using the inputted SupportPoints, their difference, and the concatenated index of both SupportPoints (calculated through bitwise operations).

Output: minkowskiPointNew returns the new MinkowskiPoint as output.

support:

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

Exceptions: There are no potential exceptions for support.

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 inputted SupportContext and the inputted Vector, and creates a new MinkowskiPoint with these SupportPoints.

Output: support returns the new MinkowskiPoint as output.

supportEdgeForPoly:

Input: supportEdgeForPoly accepts a PolyShape pointer and a Vector as inputs.

Exceptions: There are no potential exceptions for supportEdgeForPoly.

Transition: supportEdgeForPoly finds the vertices adjacent to the inputted PolyShape's support point and calculates the dot product of the vertices and the inputted Vector. The function then creates a new Edge with the support point and the vertex that maximized the product.

Output: supportEdgeForPoly returns the new Edge as output.

supportEdgeForSegment:

Input: supportEdgeForSegment accepts a SegmentShape pointer and a Vector as inputs.

Exceptions: There are no potential exceptions for supportEdgeForSegment.

Transition: supportEdgeForSegment calculates the dot product of the inputted SegmentShape's transformed normal and the inputted Vector. If the result is positive, the function creates a new Edge using the SegmentShape's endpoints and other properties. Otherwise, it creates a new Edge using the endpoints in reversed order and a negative transformed normal.

Output: supportEdgeForSegment returns the new Edge as output.

closestT:

Input: closestT accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for closestT.

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 inputted Vectors and $t \in [-1,1]$. The function clamps the result to this interval.

Output: closestT returns a double as output.

lerpT:

Input: lerpT accepts two Vectors and a double as inputs.

Exceptions: There are no potential exceptions for lerpT.

Transition: lerpT functions similarly to vectLerp, except the parameter t, the last inputted double, is constrained to the interval $[-1,1]$.

Output: lerpT returns a Vector as output.

closestPoints New:

Input: closestPointsNew accepts two MinkowskiPoint structures as inputs.

Exceptions: There are no potential exceptions for closestPointsNew.

Transition: TODO

Output: closestPointsNew returns the new ClosestPoints as output.

closestDist:

Input: closestDist accepts two Vectors as inputs.

Exceptions: There are no potential exceptions for closestDist.

Transition: closestDist calls closestT with the inputted Vectors, and uses the result to linearly interpolate the two Vectors with lerpT. The function then calculates the squared length of the resultant Vector.

Output: closestDist returns the result as a double.

EPARecurse:

Input: EPARecurse accepts a SupportContext pointer, an integer, a pointer to a MinkowskiPoint array and another integer as inputs.

Exceptions: EPARecurse may throw a SameVertices exception when the EPA vertices are the same. It may also raise HighIterWarning when the iteration number (last inputted integer) is greater than or equal to the WARN_EPA_ITERATIONS threshold.

Transition: EPARecurse is a recursive implementation of the EPA (Expanding Polytope Algorithm), where each recursion adds a point to the convex hull until the function obtains the closest point on the surface.

Output: EPARecurse returns the new ClosestPoints structure as output.

EPA:

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

Exceptions: See Exceptions for EPARecurse.

Transition: EPA finds the closest points on the surface of two overlapping Shapes using the EPA (Expanding Polytope Algorithm). The function initializes a *hull* array with the three inputted MinkowskiPoints as elements and calls EPARecurse with the inputted SupportContext, the number of elements in *hull*, the array itself and the initial iteration number (1).

Output: EPA returns the ClosestPoints structure generated by EPARecurse as output.

GJKRecurse:

Input: GJKRecurse accepts a SupportContext pointer, two MinkowskiPoint structures and an integer as inputs.

Exceptions: GJKRecurse may raise a HighIterWarning when the iteration number (last inputted integer) is greater than or equal to the WARN_GJK_ITERATIONS threshold, or WARN_EPA_ITERATIONS when EPARecurse needs to be called.

Transition: GJKRecurse is a recursive implementation of the GJK (Gilbert-Johnson-Keerthi) algorithm. If the collision Shapes are found to overlap, the function will execute EPA to find the closest points.

Output: GJKRecurse returns the new ClosestPoints structure as output.

shapePoint:

Input: shapePoint accepts a Shape pointer and an integer as inputs.

Exceptions: There are no potential exceptions for shapePoint.

Transition: shapePoint creates a new SupportPoint depending on the type of the inputted Shape and index (the inputted integer).

Output: shapePoint returns the new SupportPoint as output.

GJK:

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

Exceptions: See Exceptions for GJKRecurse.

Transition: GJK finds the closest points between two shapes using the (Gilbert-Johnson-Keerthi) algorithm.

Output: GJK returns the ClosestPoints generated by GJKRecurse as output.

contactPoints:

Input: contactPoints accepts two Edge structures, a ClosestPoints structure and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for contactPoints.

Transition: contactPoints finds contact point pairs on the surfaces of the inputted support Edges.

Output: contactPoints does not return any value.

CircleToCircle:

Input: CircleToCircle accepts two CircleShape pointers and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for CircleToCircle.

Transition: CircleToCircle checks if the current distance between the inputted CircleShapes is less than the minimum collision distance, which is the sum of the Shapes' radii. If so, the function pushes a new Contact structure to the Contacts array of the inputted CollisionInfo.

Output: CircleToCircle does not return any value.

CircleToSegment: **Input:** CircleToSegment accepts a CircleShape pointer, a SegmentShape pointer and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for CircleToSegment.

Transition: CircleToSegment finds the closest point on the inputted SegmentShape to the inputted Circle and checks if the current distance between the point and the CircleShape's center is less than the minimum collision distance, which is the sum of the Shapes' radii. If so, the function pushes a new Contact structure to the Contacts array of the inputted CollisionInfo. Coincident (completely overlapping) Shapes are taken into account and end-cap collisions are rejected if tangents are provided.

Output: CircleToSegment does not return any value.

SegmentTo Segment: **Input:** SegmentToSegment accepts two SegmentShape pointers and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for SegmentToSegment.

Transition: SegmentToSegment creates a new SupportContext structure with the inputted SegmentShapes and the segmentSupportPoint function. It then generates a ClosestPoints structure with GJK, using pointers to the context structure and the ID of the inputted CollisionInfo. Afterwards, it will check if the closest points are nearer than the minimum collision distance, which is the sum of the Shapes' radii, and if so, push a new Contact structure to the Contacts array of the inputted CollisionInfo. End-cap collisions are rejected if tangents are provided.

Output: SegmentToSegment does not return any value.

PolyToPoly: **Input:** PolyToPoly accepts two PolyShape pointers and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for PolyToPoly.

Transition: PolyToPoly creates a new SupportContext structure with the inputted PolyShapes and the polySupportPoint function. It then generates a ClosestPoints structure with GJK, using pointers to the context structure and the ID of the inputted CollisionInfo. Afterwards, it will check if the closest points are nearer than the minimum collision distance, which is the sum of the Shapes' radii, and if so, push a new Contact structure to the Contacts array of the inputted CollisionInfo.

Output: PolyToPoly does not return any value.

SegmentTo Poly:

Input: SegmentToPoly accepts a SegmentShape pointer, a PolyShape pointer and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for SegmentToPoly.

Transition: SegmentToPoly creates a new SupportContext structure with the inputted Shapes and their corresponding SupportPointFuncs. It then generates a ClosestPoints structure with GJK, using pointers to the context structure and the ID of the inputted CollisionInfo. Afterwards, it will check if the closest points are nearer than the minimum collision distance, which is the sum of the Shapes' radii, and if so, push a new Contact structure to the Contacts array of the inputted CollisionInfo. End-cap collisions are rejected if tangents are provided.

Output: SegmentToPoly does not return any value.

CircleToPoly:

Input: CircleToPoly accepts a CircleShape pointer, a PolyShape pointer and a CollisionInfo pointer as inputs.

Exceptions: There are no potential exceptions for CircleToPoly.

Transition: CircleToPoly creates a new SupportContext structure with the inputted Shapes and their corresponding SupportPointFuncs. It then generates a ClosestPoints structure with GJK, using pointers to the context structure and the ID of the inputted CollisionInfo. Afterwards, it will check if the closest points are nearer than the minimum collision distance, which is the sum of the Shapes' radii, and if so, push a new Contact structure to the Contacts array of the inputted CollisionInfo.

Output: CircleToPoly does not return any value.

CollisionError: **Input:** CollisionError accepts two Shape pointers and a CollisionInfo pointer as inputs.

Exceptions: CollisionError throws an eponymous exception when the types of the inputted Shapes are not in sorted order.

Transition: CollisionError throws an exception and aborts the program. This function is called by collide when the colliding Shape types are not in order.

Output: CollisionError does not return any value.

14 MIS of the Sequence Data Structure Module

14.1 Module Name: Array

14.2 Uses

This module only uses C's 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*	-
arrayDeleteObj	Array*, void*	-	-
arrayContains	Array*, void*	Boolean	-
arrayFreeEach	Array*, void* → 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 inputted pointers are assumed to be non-null.

14.4.4 Access Program Semantics

arrayNew:

Input: arrayNew accepts an integer as input.

Exceptions: There are no potential exceptions for arrayNew.

Transition: arrayNew heap-allocates a new Array object. It sets the Array's length and maximum length to the inputted integer (unless the input is zero, in which case the default maximum is 4), and heap-allocate 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: There are no potential exceptions for arrayFree.

Transition: arrayFree frees the internal array of the inputted Array, and then frees the Array itself.

Output: arrayFree does not return any value.

arrayPush: **Input:** arrayPush accepts an Array pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for arrayPush.

Transition: arrayPush inserts the specified element into the inputted 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: arrayPush does not return any value.

arrayPop: **Input:** arrayPop accepts an Array pointer as input.

Exceptions: There are no potential exceptions for arrayPop.

Transition: arrayPop will remove the last element of the inputted 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: There are no potential exceptions for arrayDeleteObj.

Transition: arrayDeleteObj deletes the specified element (void pointer) from the inputted Array and decrements the number of elements accordingly.

Output: arrayDeleteObj does not return any value.

arrayContains: **Input:** arrayContains accepts an Array pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for arrayContains.

Transition: arrayContains iterates through the inputted Array and attempts to find the inputted element (void pointer) in the Array.

Output: arrayContains returns true if the inputted Array contains the inputted 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* → void).

Exceptions: There are no potential exceptions for arrayFreeEach.

Transition: arrayFreeEach iterates through the internal array of the inputted Array and applies the inputted function to each element.

Output: arrayFreeEach does not return any value.

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*	-

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 inputted pointers are assumed to be non-null.

15.4.3 Access Program Semantics

BBTreeAlloc:

Input: BBTreeAlloc does not accept any inputs.

Exceptions: There are no potential exceptions for BBTreeAlloc.

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: There are no potential exceptions for BBTreeInit.

Transition: BBTreeInit initializes the inputted BBTree as a SpatialIndex, using the BBTree SpatialIndexClass klass and the inputted 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 warning if the inputted SpatialIndex is not a BBTree.

Transition: BBTreeSetVelocityFunc sets the inputted BBTree SpatialIndex's internal velocity function to the provided function.

Output: BBTreeSetVelocityFunc does not return any value.

BBTreeNew: **Input:** BBTreeNew accepts a SpatialIndexBBFunc function pointer and a SpatialIndex pointer as inputs.

Exceptions: There are no potential exceptions for BBTreeNew.

Transition: BBTreeNew allocates a new BBTree from the heap and initializes it.

Output: BBTreeNew returns a pointer to the initialized BBTree.

15.4.4 Local Constants

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

15.4.5 Local Functions

getBB: **Input:** getBB accepts a BBTree pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for getBB.

Transition: getBB retrieves the BB corresponding to the inputted object (void pointer) from the inputted BBTree. If the tree has a valid velocity function, the boundaries of the BB will be adjusted accordingly.

Output: getBB returns the retrieved BB as output.

getTree: **Input:** getTree accepts a SpatialIndex pointer as input.

Exceptions: There are no potential exceptions for getTree.

Transition: If the inputted SpatialIndex is of the BBTree class, the function returns the inputted pointer cast as a BBTree pointer. Otherwise, it returns a null pointer..

Output: getTree returns a BBTree pointer as output.

getRootIfTree: **Input:** getRootIfTree accepts a SpatialIndex pointer as input.

Exceptions: There are no potential exceptions for getRootIfTree.

Transition: If the inputted SpatialIndex is of the BBTree class, the function casts the inputted pointer to a BBTree pointer and returns a Node pointer to its root. Otherwise, it returns a null pointer.

Output: getRootIfTree returns a Node pointer as output.

getMasterTree: **Input:** getMasterTree accepts a BBTree pointer as input.

Exceptions: There are no potential exceptions for getMasterTree.

Transition: getMasterTree attempts to retrieve the BBTree's master tree, which is its index of dynamic bodies.

Output: If the function retrieves a valid tree, it returns the tree. Otherwise, it returns the inputted tree.

incrementStamp: **Input:** incrementStamp accepts a BBTree pointer as input.

Exceptions: There are no potential exceptions for incrementStamp.

Transition: incrementStamp attempts to retrieve a BBTree's master tree. If the retrieved tree is valid, the function increments the timestamp of this tree. Otherwise, it will increment the timestamp of the inputted tree.

Output: incrementStamp does not return any value.

pairRecycle: **Input:** pairRecycle accepts a BBTree pointer and a Pair pointer as inputs.

Exceptions: There are no potential exceptions for pairRecycle.

Transition: pairRecycle retrieves the master tree of the inputted BBTree and recycles the inputted Pair to the tree's pooled Pairs.

Output: pairRecycle does not return any value.

pairFromPool: **Input:** pairFromPool accepts a BBTree pointer as input.

Exceptions: pairFromPool may throw an InsufficientBufferSize exception if the size of a Pair object exceeds the buffer size (BUFFER_BYTES).

Transition: pairFromPool retrieves the master tree of the inputted BBTree and retrieves a Pair from the inputted BBTree's pooled Pairs. If there is none, the function allocates a new Pair buffer, adds it to the tree's allocated buffers, and adds all the new Pairs to the pool except the first one, which is returned.

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

threadUnlink:

Input: threadUnlink accepts a Thread as input.

Exceptions: There are no potential exceptions for threadUnlink.

Transition: threadUnlink removes the inputted Thread from the chain.

Output: threadUnlink does not return any value.

pairsClear:

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

Exceptions: There are no potential exceptions for pairsClear.

Transition: pairsClear removes all Pairs associated with the inputted Node, unlinks all Threads associated with each Pair and recycles the Pairs.

Output: pairsClear does not return any value.

pairInsert:

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

Exceptions: There are no potential exceptions for pairInsert.

Transition: pairInsert obtains a pooled Pair from the inputted BBTree, creates a new Pair with the inputted Nodes and inserts the Pair by linking it with the Threads associated with the Nodes.

Output: pairInsert does not return any value.

nodeRecycle:

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

Exceptions: There are no potential exceptions for nodeRecycle.

Transition: nodeRecycle removes the inputted Node from the inputted BBTree and recycles it to the tree's pooled Nodes.

Output: nodeRecycle does not return any value.

nodeFromPool: **Input:** nodeFromPool accepts a BBTree pointer as input.

Exceptions: nodeFromPool may throw an InsufficientBufferSize exception if the size of a Node object exceeds the buffer size (BUFFER_BYTES).

Transition: nodeFromPool obtains a Node from the inputted BBTree's pooled Nodes. If there are none, the function allocates a new Node buffer, adds it to the tree's allocated buffers, and adds all the new Nodes to the pool except for the first one, which is returned.

Output: nodeFromPool returns the retrieved Node as output.

nodeSet: **Input:** Each nodeSet function accepts two Node pointers as input.

Exceptions: There are no potential exceptions for each nodeSet function.

Transition: Each nodeSet function sets the corresponding children of the first inputted Node to the second inputted Node, and the second Node's parent to the first Node.

Output: Each nodeSet function does not return any value.

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

Exceptions: There are no potential exceptions for nodeNew.

Transition: nodeNew retrieves a pooled Node and initializes it. The function sets this Node's children to the two inputted Nodes, its BB to the merged BBs of the two Nodes, and all other values to null.

Output: nodeNew returns the initialized Node as output.

nodeIsLeaf: **Input:** nodeIsLeaf accepts a Node pointer as input.

Exceptions: There are no potential exceptions for nodeIsLeaf.

Transition: nodeIsLeaf checks if the inputted Node is a leaf.

Output: nodeIsLeaf returns true if the inputted Node has a valid non-null object, and false otherwise.

nodeOther:

Input: nodeOther accepts two Node pointers as inputs.

Exceptions: There are no potential exceptions for nodeOther.

Transition: nodeOther retrieves the sibling of the second inputted Node.

Output: nodeOther returns the sibling Node as output.

nodeReplace Child:

Input: nodeReplaceChild accepts three Node pointers and a BB-Tree 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 inputted Node) is not a child of the parent Node (first inputted Node).

Transition: nodeReplaceChild replaces the child (second inputted Node) of the parent Node (first inputted Node) with the third inputted Node, and updates the BBs of all parents of the parent Node.

Output: nodeReplaceChild does not return any value.

BBProximity:

Input: BBProximity accepts two BBs as inputs.

Exceptions: There are no potential exceptions for BBProximity.

Transition: BBProximity calculates the proximity of the inputted BBs to each other.

Output: BBProximity returns the result as a double.

subtreeInsert:

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

Exceptions: There are no potential exceptions for subtreeInsert.

Transition: subtreeInsert inserts the second inputted Node into the subtree originating from the first inputted Node, and recalculates the BB of this root.

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: There are no potential exceptions for subtreeRemove.

Transition: subtreeRemove deletes the second inputted Node and its parent from the subtree originating from the first inputted Node.

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: There are no potential exceptions for markLeafQuery.

Transition: markLeafQuery only makes transitions if the BBs of the two inputted Nodes intersect. In that case, the function will check if the first inputted Node is a leaf. If not, the function will recursively search through the left and right subtrees of the first inputted Node. Otherwise, and if the Node is a left child (the inputted Boolean value is true), a new Pair will be added with the two Nodes. If the Node is a right child (the inputted 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 inputted MarkContext with the objects of both Nodes, the ID zero, and the data (void pointer) in the context.

Output: markLeafQuery does not return any value.

markLeaf: **Input:** markLeaf accepts a Node pointer and a MarkContext pointer as inputs.

Exceptions: There are no potential exceptions for markLeaf.

Transition: markLeaf checks if first inputted Node was last updated at the same time as the master tree. If this is true, and if the inputted MarkContext has a valid static root, the function will call markLeafQuery with the root, the inputted 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 inputted 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.

Output: markLeaf does not return any value.

markSubtree:

Input: markSubtree accepts a Node pointer and a MarkContext pointer as inputs.

Exceptions: There are no potential exceptions for markSubtree.

Transition: markSubtree checks if the inputted Node is a leaf. If so, it will mark the Node. Otherwise, markSubtree will recursively find and mark its children.

Output: markSubtree does not return any value.

leafNew:

Input: leafNew accepts a BBTree pointer, a void pointer and a BB as inputs.

Exceptions: There are no potential exceptions for leafNew.

Transition: leafNew retrieves a pooled Node from the inputted Tree and initializes the Node with the inputted object (void pointer) and BB. All other variables are zero-initialized.

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: There are no potential exceptions for each leafUpdate function.

Transition: leafUpdate obtains the BB corresponding to the object of the first inputted 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.

Output: If the Node is updated, leafUpdate returns true. Otherwise, it returns false. leafUpdateWrap calls this function, but discards the Boolean output.

voidQueryFunc: **Input:** voidQueryFunc accepts two void pointers, a CollisionID and another void pointer as inputs.

Exceptions: There are no potential exceptions for voidQueryFunc.

Transition: voidQueryFunc makes no transition.

Output: voidQueryFunc returns the inputted CollisionID as output.

leafAddPairs: **Input:** leafAddPairs accepts a Node pointer and a BBTree pointer as inputs.

Exceptions: There are no potential exceptions for leafAddPairs.

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 inputted 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 inputted Tree, the root, and voidQueryFunc, and mark the inputted Node using this context.

Output: leafAddPairs does not return any value.

leafSetEql: **Input:** leafSetEql accepts a void pointer and a Node pointer as inputs.

Exceptions: There are no potential exceptions for leafSetEql.

Transition: leafSetEql checks if the inputted object (void pointer) is equal to the inputted Node's object.

Output: leafSetEql returns the Boolean result of the above test.

leafSetTrans: **Input:** leafSetTrans accepts a void pointer and a BBTree pointer as inputs.

Exceptions: There are no potential exceptions for leafSetTrans.

Transition: leafSetTrans creates a new leaf Node with the inputted Tree, object (void pointer) and BB corresponding to the object.

Output: leafSetTrans returns a void pointer to a new leaf Node as output.

BBTreeDestroy: **Input:** BBTreeDestroy accepts a BBTree pointer as input.

Exceptions: There are no potential exceptions for BBTreeDestroy.

Transition: BBTreeDestroy frees the dynamically-allocated structures of the inputted BBTree and all of its elements, namely its leaves and allocated buffers.

Output: BBTreeDestroy does not return any value.

BBTreeCount: **Input:** BBTreeCount accepts a BBTree pointer as input.

Exceptions: There are no potential exceptions for BBTreeCount.

Transition: BBTreeCount counts the number of leaves contained in the inputted Tree.

Output: BBTreeCount returns the result as an integer.

eachHelper: **Input:** eachHelper accepts a Node pointer and an EachContext pointer as inputs.

Exceptions: There are no potential exceptions for eachHelper.

Transition: eachHelper calls the function contained in the inputted EachContext with the inputted Node's object and the context's data.

Output: eachHelper does not return any value.

BBTreeEach: **Input:** BBTreeEach accepts a BBTree pointer, a SpatialIndexIteratorFunc function pointer, and a void pointer as inputs.

Exceptions: There are no potential exceptions for BBTreeEach.

Transition: BBTreeEach creates a new EachContext structure with the inputted function and data (void pointer), and iterates through the inputted BBTree's leaves using the hash set iterator, eachHelper and the context structure.

Output: BBTreeEach does not return any value.

BBTreeInsert: **Input:** BBTreeInsert accepts a BBTree pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for BBTreeInsert.

Transition: BBTreeInsert inserts a new Node with the inputted object (void pointer) and HashValue into the inputted 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 inputted tree.

Output: BBTreeInsert does not return any value.

BBTreeRemove: **Input:** BBTreeRemove accepts a BBTree pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for BBTreeRemove.

Transition: BBTreeRemove deletes the Node corresponding to the inputted object (void pointer) and HashValue from the inputted BBTree, clears the Pairs for that Node and recycles the empty Node.

Output: BBTreeRemove does not return any value.

BBTreeContains: **Input:** BBTreeContains accepts a BBTree pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for BBTreeContains.

Transition: BBTreeContains searches the leaves of the inputted BB-Tree for the inputted object (void pointer) and HashValue.

Output: BBTreeContains 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: There are no potential exceptions for BBTreeReindexQuery.

Transition: If the inputted 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 inputted 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 inputted parameters. Finally, the function updates the timestamp of the tree.

Output: BBTreeReindexQuery does not return any value.

BBTreeReindex: **Input:** BBTreeReindex accepts a BBTree pointer as input.

Exceptions: There are no potential exceptions for BBTreeReindex.

Transition: BBTreeReindex calls BBTreeReindexQuery with the inputted BBTree, voidQueryFunc and a null pointer.

Output: BBTreeReindex does not return any value.

BBTreeReindex Object: **Input:** BBTreeReindexObject accepts a BBTree pointer, a void pointer and a HashValue as inputs.

Exceptions: There are no potential exceptions for BBTreeReindexObject.

Transition: The function will attempt to find the Node corresponding to the inputted 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: BBTreeReindexObject does not return any value.

Klass: **Input:** Klass does not accept any input.

Exceptions: There are no potential exceptions for Klass.

Transition: Klass makes no transition.

Output: Klass returns a pointer to the SpatialIndexClass klass.

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

16.4.2 State Invariant

HashSet.entries $\leq$ HashSet.size

16.4.3 Assumptions

hashSetNew is called before any other access programs, and all inputted 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: There are no potential exceptions for HASH_PAIR.

Transition: HASH_PAIR calculates a new HashValue from the pair of inputted HashValues.

Output: HASH_PAIR returns the new HashValue as output.

hashSetNew:

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

Exceptions: There are no potential exceptions for hashSetNew.

Transition: hashSetNew heap-allocates a new HashSet, where the size is the next prime number greater than the inputted integer. The HashSet's internal equality function is set to the inputted 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.

hashSetSetDefaultValue:

Input: hashSetSetDefaultValue accepts a HashSet pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for hashSetSetDefaultValue.

Transition: hashSetSetDefaultValue sets the inputted HashSet's default value variable to the inputted void pointer.

Output: hashSetSetDefaultValue does not return any value.

hashSetFree:

Input: hashSetFree accepts a HashSet pointer as input.

Exceptions: There are no potential exceptions for hashSetFree.

Transition: hashSetFree frees the internal hash table of the inputted HashSet, frees its allocated buffers, and finally frees the HashSet itself.

Output: hashSetFree does not return any value.

hashSetCount:

Input: hashSetCount accepts a HashSet pointer as input.

Exceptions: There are no potential exceptions for hashSetCount.

Transition: hashSetCount makes no transition.

Output: hashSetCount returns the number of hash table entries contained in the inputted 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: There are no potential exceptions for hashSetInsert.

Transition: hashSetInsert inserts the inputted element (first void pointer) into the inputted 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: There are no potential exceptions for hashSetRemove.

Transition: hashSetRemove deletes the inputted element (void pointer) from the inputted 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: There are no potential exceptions for hashSetFind.

Transition: hashSetFind searches through the inputted HashSet and attempts to find the inputted element (void pointer).

Output: 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: There are no potential exceptions for hashSetEach.

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

Output: hashSetEach does not return any value.

hashSetFilter:

Input: hashSetFilter accepts a HashSet pointer, a HashSetFilterFunc function pointer and a void pointer as inputs.

Exceptions: There are no potential exceptions for hashSetFilter.

Transition: hashSetFilter iterates through the entries of the inputted HashSet and removes all entries for which the inputted filtering function returns false. For each removed element, the bin containing the entry is recycled. The number of entries is updated accordingly.

Output: hashSetFilter does not return any value.

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 inputted integer is greater than 1610612741.

Transition: next_prime iterates through the primes array and finds the nearest prime that is greater than the inputted integer.

Output: next_prime returns the next greatest prime as output.

setIsFull:

Input: setIsFull accepts a HashSet pointer as input.

Exceptions: There are no potential exceptions for setIsFull.

Transition: setIsFull checks if the inputted HashSet is at capacity.

Output: setIsFull returns the Boolean result of the above test as output.

hashSetResize: **Input:** hashSetResize accepts a HashSet pointer as input.

Exceptions: There are no potential exceptions for hashSetResize.

Transition: hashSetResize allocates a new hash table for the inputted 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: hashSetResize does not return any value.

recycleBin: **Input:** recycleBin accepts a HashSet pointer and a HashSetBin pointer as inputs.

Exceptions: There are no potential exceptions for recycleBin.

Transition: recycleBin deletes the element of the inputted HashSetBin and adds it to the inputted HashSet's pooled bins.

Output: recycleBin does not return any value.

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 inputted 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.

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.”
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.”