

Module Interface Specification for Tamias2D : A 2D Game Physics Library

Luthfi Mawarid and Oluwaseun Owojaiye

December 28, 2018

1 Revision History

Date	Version	Notes
Nov. 15, 2018	1.0	Updates throughout document; siimplifying document
Nov 20, 2018	1.1	Notes

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at <https://github.com/smiths/caseStudies/blob/master/CaseStudies/gamephys/docs/SRS/GamePhysicsSRS.pdf>

Contents

1	Revision History	i
2	Symbols, Abbreviations and Acronyms	ii
3	Introduction	1
4	Notation	1
5	Module Decomposition	2
6	MIS of the Rigid Body Module	4
6.1	Module	4
6.2	Uses	4
6.3	Syntax	4
6.3.1	Exported Constants	4
6.3.2	Exported Access Programs	4
6.3.3	Exported Constants	4
6.4	Semantics	4
6.4.1	State Variables	4
6.4.2	Environment Variables	5
6.4.3	Assumptions	5
6.4.4	Access Routine Semantics	5
6.4.5	Local Functions	6
7	MIS of the Shape Module	6
7.1	Module Name: Shape	6
7.2	Uses	6
7.3	Syntax	7
7.3.1	Exported Constants	7
7.3.2	Exported Access Programs	7
7.4	Semantics	7
7.4.1	State Variables	7
7.4.2	Environment Variables	7
7.4.3	Assumptions	7
7.4.4	Access Program Semantics	7
7.5	Submodule Name: CircleShape	8
7.6	Uses	8
7.7	Syntax	8
7.7.1	Exported Data Types	8
7.7.2	Exported Access Programs	8
7.8	Semantics	9
7.8.1	State Variables	9

7.8.2	Assumptions	9
7.8.3	Access Program Semantics	9
7.8.4	Local Constants	10
7.8.5	Local Functions	11
7.9	Submodule Name: PolyShape	11
7.10	Uses	11
7.11	Syntax	11
7.11.1	Exported Data Types	11
7.11.2	Exported Access Programs	11
7.12	Semantics	12
7.12.1	State Variables	12
7.12.2	Assumptions	13
7.12.3	Semantics	13
7.12.4	Local Constants	16
7.12.5	Local Functions	16
8	MIS of the Space Module	17
8.1	Module Name: Space	17
8.2	Uses	17
8.3	Syntax	17
8.3.1	Exported Constants	17
8.3.2	Exported Access Programs	17
8.4	Semantics	17
8.4.1	State Variables	17
8.4.2	Assumptions	18
8.4.3	Semantics	18
8.4.4	Local Functions	19
9	MIS of the Vector Module	21
9.1	Module Name: Vector	21
9.2	Uses	21
9.3	Syntax	21
9.3.1	Exported Constants	21
9.3.2	Exported Access Programs	21
9.4	Semantics	21
9.4.1	State Variables	21
9.4.2	Semantics	22
10	MIS of the Collision Solver Module	24
10.1	Module Name: Collision	24
10.2	Uses	24
10.3	Syntax	24
10.3.1	Exported Constants	24

10.3.2	Exported Access Programs	24
10.4	Semantics	25
10.4.1	State Variables	25
10.4.2	Access Program Semantics	26
10.4.3	Local Constants	28
10.4.4	Local Functions	28

3 Introduction

The following document details the Module Interface Specifications for the implemented modules in the Tamias2D 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. The full documentation and implementation can be found at <https://github.com/smiths/caseStudies/tree/master/CaseStudies/gamephys>

4 Notation

The structure of the MIS for modules comes from ? , with the addition that template modules have been adapted from ? . The mathematical notation comes from Chapter 3 of ? . For instance, the symbol $:=$ is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | \dots | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Tamias2D .

Data Type	Notation	Description
boolean	\mathbb{B}	An element of $\{\text{true}, \text{false}\}$.
character	char	A single symbol or digit.
real	\mathbb{R}	Any number in $(-\infty, \infty)$.
integer	\mathbb{Z}	A number without a fractional component in $(-\infty, \infty)$.
natural number	\mathbb{N}	A number without a fractional component in $[1, \infty)$.

The specification of Tamias2D uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Tamias2D uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

Here is a table to summarize the derived data types:

Data Type	Notation	Description
bodyType	Body	an object used by Tamias2D to represent a rigid body
vector	Vec2d	an object used by Tamias2D to store vector values i.e x, y coordinates
space	Space	an object used by Tamias2D to represent the simulation space

[I'm not sure we need pointers at this point. An abstract design probably shouldn't have to invoke the concept of pointers. —SS][updated —OO]

[I really don't think we need void in the abstract MIS. If an access program doesn't return any value, we just don't assign it one. —SS] [removed section —OO]

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	Level 2	Level 3
Hardware-Hiding Module		
	Rigid Body	
Behaviour-Hiding Module	Shape	
		Circle Polygon
	Space	
Software Decision Module	Vector Collision Solver	

Table 2: Module Hierarchy

[put a newpage between all modules. —SS][noted —OO]

6 MIS of the Rigid Body Module

6.1 Module

Body

6.2 Uses

Shape Module, Space Module, Vector Module

6.3 Syntax

6.3.1 Exported Constants

N/A

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
BodyInit	$\mathbb{R}, \mathbb{R}, (\mathbb{R}_x, \mathbb{R}_y), \mathbb{R}, \mathbb{N}, \mathbb{R}$	<i>Body</i>	massError
Update	<i>Body</i>	$(\mathbb{R}_x, \mathbb{R}_y), (\mathbb{R}_x, \mathbb{R}_y)$	-
apply_force	<i>Body</i>	<i>Vec2d</i>	-
apply_torque	<i>Body</i>	\mathbb{R}	-
set_space	<i>Body</i>	<i>Space</i>	-
set_angular_accler	<i>Body</i>	\mathbb{R}	-

6.3.3 Exported Constants

$C_R := 1$

6.4 Semantics

6.4.1 State Variables

[if you are going to create a new type, then this should be an exported type, not a state variable. —SS]

[I don't know what this heading means? You can just have your list of state variables. —SS]

type: BodyType	position: Vec2d	torque: \mathbb{R}
mass: \mathbb{R}	velocity: Vec2d	space: Space
massInv: \mathbb{R}	force: Vec2d	
moment: \mathbb{R}	angle: \mathbb{R}	
momentInv: \mathbb{R}	angular_accl: \mathbb{R}	

[You don't need the pointers. That is an implementation detail. —SS] [updated section —OO]

[Most of the state invariants above are just restating the type of the state variables. This is redundant and can be removed. For more interesting invariants, like the one about the mass being positive, you need to make sure that the constructor will not allow this invariant to be violated. I can see below that this is not done. Invariants aren't automatically true. Your specification needs to enforce them. —SS]

6.4.2 Environment Variables

N/A

6.4.3 Assumptions

N/A [This is far too implementation specific. Don't worry about pointers at this point. —SS]

6.4.4 Access Routine Semantics

bodyInit:	Transition:	bodyInit allocates a new Body and initializes its mass, moment, bodyType, restitution, angle and shape with the input values. Other fields are zero-initialized and kinematic functions are set to default ones. [Where do these inputs come from? Only two real numbers are passed to body? —SS]
	Output:	bodyInit returns a pointer to the initialized Body.
	Exceptions:	None. [Why aren't you checking that the mass is positive? —SS]
Update :	Transition:	Update will update Body's position and all other parameters (bodyInit) associated to the body after each cycle. [How is the update done? The equations from your SRS should appear in your MIS. You haven't given enough information for someone to code this module. —SS]
	Output:	None. [If there is no output, do not include this field. Remember that output and transition are unlikely to go together, except for constructors. —SS]
	Exceptions:	None
apply_force:	Transition:	apply_force will apply forces on the rigid bodies and it recalculates the total force acting on each body. [What state variable is updated? —SS]

	Output:	None.
	Exceptions:	None
apply_torque:	Transition:	apply_torque will apply forces(for rotation) on rigid bodies and recalculates the total torque acting on each body. [What state variable is updated? How is it changed? I should see an explicit change to a state variable, not just vague words. —SS]
	Output:	None.
	Exceptions:	None

[There are state variables that do not appear to have any transitions. How are they influenced? Is there no way to inspect the values of the state variables? You have an access program for set space, but you don't actually specify it. —SS]

6.4.5 Local Functions

N/A

7 MIS of the Shape Module

[I have similar comments on this module to the previous. Exported types should be in an exported types field, or in their own ADT. You shouldn't show transition and output for every module. You don't want to use pointers. Put a newpage between modules. Transitions and outputs should be expressed as equations, not words. —SS]

[Modifying the previous documentation by deleting the things that don't apply isn't really working well. The previous design is far too tied to the Chipmunk implementation. We want the new documentation to follow the MIS examples we showed in class. We also want it much simpler. It should be based on the SRS, not the code. —SS]

[You seem to have inheritance, but you haven't used the way we covered in class to show it. —SS]

7.1 Module Name: Shape

7.2 Uses

Rigid Body Module, Space Module, Vector Module,

7.3 Syntax

7.3.1 Exported Constants

N/A

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
set_angle	Shape*	\mathbb{R}	-
set_pos	Shape*	Vec2*	-
get_center	Shape*	Vec2*	-

7.4 Semantics

7.4.1 State Variables

ShapeType $\in \{\text{CIRCLE}, \text{POLYGON}\}$

ShapeClass:

type: ShapeType

Shape:

Colshape: ShapeClass*

type: CollisionType

space: Space*

body: Body*

7.4.2 Enviroment Vaiables

None

7.4.3 Assumptions

All input pointers are assumed to be non-null. Also see [7.8.2](#), [7.12.2](#).

7.4.4 Access Program Semantics

	Transition:	None.
get_center:	Exceptions:	None.
	Output:	get_center returns the coordinates of the input Shape object.

set_angle: **Transition:** Each set_angle function sets their corresponding angle with the input value

Exceptions: None.

Output: None.

set_pos: **Transition:** Each set_pos function sets their corresponding position of a shape pointer in space

Exceptions: None.

Output: None.

7.5 Submodule Name: CircleShape

[Out template doesn't have a heading called Submodule. I don't know what this means. Document inheritance as we discussed it in class. —SS]

7.6 Uses

Rigid Body Module, Shape Module, Vector Module

7.7 Syntax

7.7.1 Exported Data Types

CircleShape: struct

7.7.2 Exported Access Programs

Name	In	Out	Exceptions
circleShapeInit	CircleShape*, Body*, double, Vector	CircleShape*	-
get_center	Shape*	double	NotCircleShape
circleShapeNew	Body*, double, Vector	Shape*	-
circleShapeGetRadius	Shape*	double	NotCircleShape
circleShapeSetRadius	Shape*, double	-	NotCircleShape ∨ IllegalBody
momentForCircle	double, double, double, Vector	double	-

areaForCircle	double, double	double	-
---------------	----------------	--------	---

[How is the center of the circle a double? —SS] [Why aren't the semantics given for all of the access programs? —SS]

7.8 Semantics

7.8.1 State Variables

CircleShape:

shape: Shape

center: Vector

radius: \mathbb{R}

7.8.2 Assumptions

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

7.8.3 Access Program Semantics

circleShapeInit:	Input:	circleShapeInit accepts a CircleShape pointer, a Body pointer, a double and a Vector as inputs.
	Exceptions:	None.
	Transition:	circleShapeInit initializes the input CircleShape. It sets the radius to the input double, the center to the input Vector, and then initializes the rest of the variables using shapeInit and the input Body.
	Output:	circleShapeInit returns a pointer to the initialized CircleShape.
circleShapeNew:	Input:	circleShapeNew accepts a Body pointer, a double and a Vector as inputs.
	Exceptions:	None.
	Transition:	circleShapeNew allocates and initializes a new CircleShape object using the input parameters.
	Output:	circleShapeNew returns a pointer to the new CircleShape.
circleShapeGet:	Input:	Each circleShapeGet function accepts a Shape pointer as input.

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

7.8.4 Local Constants

CircleShapeClass: ShapeClass

CircleShapeClass := {CIRCLE_SHAPE, circleShapeCacheData, NULL}

7.8.5 Local Functions

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

Exceptions: None.

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

Output: circleShapeCacheData returns the new BB as output.

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

Exceptions: None.

Transition: None.

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

7.9 Submodule Name: PolyShape

7.10 Uses

[Rigid Body Module](#), [Shape Module](#), [Vector Module](#),

7.11 Syntax

7.11.1 Exported Data Types

PolyShape: struct

7.11.2 Exported Access Programs

Name	In	Out	Exceptions
polyShapeInit	PolyShape*, Body*, int, Vector*, double, Transform	PolyShape*	-

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

7.12 Semantics

7.12.1 State Variables

PolyShape:

shape: Shape

count: \mathbb{Z}

radius: \mathbb{R}

7.12.2 Assumptions

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

7.12.3 Semantics

polyShapeInit:	Input:	polyShapeInit accepts a PolyShape pointer, a Body pointer, an integer, a pointer to a Vector array, a double and a Transform matrix as inputs.
	Exceptions:	None.
	Transition:	polyShapeInit transforms each vertex from the input array with the input Transform matrix, places the resultant vertices in a new array, calculates the size of the convex hull containing the new vertices and initializes the input PolyShape using this array, the hull size and the remaining parameters.
	Output:	polyShapeInit returns a pointer to the initialized PolyShape.
polyShapeInitRaw:	Input:	polyShapeInitRaw accepts a PolyShape pointer, a Body pointer, an integer, a pointer to a Vector array and a double as inputs.
	Exceptions:	None.
	Transition:	polyShapeInitRaw initializes the input PolyShape using shapeInit and the input parameters, sets its vertices to the given array and integer (which represents the length of the array), and sets its radius to the input double.
	Output:	polyShapeInitRaw returns a pointer to the initialized PolyShape.
boxShapeInit:	Input:	boxShapeInit accepts a PolyShape pointer, Body pointer and three doubles as inputs.
	Exceptions:	None.
	Transition:	boxShapeInit calculates values for half-width and half-height using the last two input doubles as width and height, respectively. It then initializes the input PolyShape using a new BB generated from the calculated half-dimensions and the remaining parameters.
	Output:	boxShapeInit returns a pointer to the initialized PolyShape.

boxShapeInit2:	Input:	boxShapeInit2 accepts a PolyShape pointer, Body pointer, a double and a BB as inputs.
	Exceptions:	None.
	Transition:	boxShapeInit2 creates a Vector array containing the vertices of the box, determined from the input BB. It then initializes the input PolyShape as a box using the array and number of vertices, as well as the remaining parameters.
	Output:	boxShapeInit2 returns a pointer to the initialized PolyShape.
polyShapeNew:	Input:	Each polyShapeNew function accepts a Body pointer, an integer, a pointer to a Vector array and a double as inputs. In addition, polyShapeNew (not Raw) accepts a Transform matrix as its last input.
	Exceptions:	None.
	Transition:	Each polyShapeNew function allocates and initializes a new PolyShape object using the input parameters.
	Output:	Each polyShapeNew function returns a pointer to the new PolyShape.
boxShapeNew:	Input:	Each boxShapeNew function accepts a Body pointer and a double as inputs. In addition, boxShapeNew accepts two additional doubles, while boxShapeNew2 accepts an additional BB as input.
	Exceptions:	None.
	Transition:	Each boxShapeNew function allocates and initializes a new PolyShape object as a box using the input parameters.
	Output:	Each boxShapeNew function returns a pointer to the new PolyShape.
polyShapeGet:	Input:	Each polyShapeGet function accepts a Shape pointer as input. polyShapeGetVert also accepts an additional integer as input.
	Exceptions:	Each polyShapeGet function may throw a NotPolyShape exception if the input Shape pointer is not of the PolyShape class. polyShapeGetVert may also throw an exception if the input integer is greater than or equal to the number of vertices of the input Shape.
	Transition:	None.

	Output:	Each polyShapeGet function returns the value of their corresponding parameter.
polyShapeSet:	Input:	Each polyShapeSet function accepts a Shape pointer and their corresponding parameter as inputs. Specifically, each polyShapeSetVerts function accepts an integer (for the number of vertices) and a pointer to a Vector array (holding the vertices) as inputs, and polyShapeSetVerts (not Raw) accepts an additional Transform matrix.
	Exceptions:	Each polyShapeSet function may throw a NotPolyShape exception if the input Shape pointer is not of the PolyShape class. Each polyShapeSetVerts function may throw an IllegalBody exception if the Body associated with the Shape violates an invariant in ?? after the transitions are complete.
	Transition:	Each polyShapeSet function sets their corresponding parameter with the input value. More specifically, polyShapeVerts transforms the vertices in the input array with the input Transform matrix, places the resultant vertices in a new array, determines the size of the convex hull containing these vertices, and calls polyShapeSetVertsRaw with the new array and hull size. polyShapeVertsRaw frees the current vertices of the input PolyShape, sets its new vertices, updates the mass information of the Shape and recalculates the mass of the associated Body.
	Output:	None.
momentForPoly:	Input:	momentForPoly accepts a double for mass, an integer for number of vertices, a pointer to a Vector array containing these vertices, a Vector for offset, and a double for radius as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	momentForPoly returns the calculated moment from the input parameters as a double.
areaForPoly:	Input:	areaForPoly accepts an integer for number of vertices, a pointer to a Vector array containing these vertices, and a double for radius as inputs.
	Exceptions:	None.

	Transition:	None.
	Output:	areaForPoly returns the calculated area from the input parameters as a double.
centroidForPoly:	Input:	centroidForPoly accepts an integer for number of vertices and a pointer to a Vector array containing these vertices as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	centroidForPoly returns the calculated centroid from the input parameters as a Vector.

7.12.4 Local Constants

PolyShapeClass: ShapeClass

PolyShapeClass := {POLY_SHAPE, polyShapeCacheData, polyShapeDestroy}

7.12.5 Local Functions

setVerts:	Input:	setVerts accepts a PolyShape pointer, an integer, and a pointer to a Vector array as inputs.
	Exceptions:	None.
	Transition:	setVerts sets the input PolyShape's number of vertices to the input integer. If this is less than or equal to POLY_SHAPE_INLINE_ALLOC , the PolyShape uses its default <code>_planes</code> array for its vertices. Otherwise, it heap-allocates a new array with the length of the input integer. Finally, the function iterates through the <code>planes</code> array and sets the vertices and their calculated edge normals from the input array. Called by polyShapeInitRaw and polyShapeSetVertsRaw to mutate vertices.
	Output:	None.

8 MIS of the Space Module

8.1 Module Name: Space

8.2 Uses

Rigid Body Module, Shape Module, Vector Module, Collision Solver Module

8.3 Syntax

8.3.1 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)

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
spaceInit	Space*	Space*	-
spaceSetGravity	Space*, Vector	-	-
spaceGetBodies	Space*	array*	-
spaceAddCollisionHandler	Space*, CollisionType, CollisionType	CollisionHandler*	-
spaceAddBody	Space*, Body*	Body*	-
spaceCollideShapes	Shape*, Shape*, CollisionID, Space*	CollisionID	-
spaceStep	Space*, double	-	-

8.4 Semantics

8.4.1 State Variables

Space:

iterations: \mathbb{Z}

gravity: Vector

Bodies: Array*

stamp: Timestamp

CollisionHandler:

typeA: CollisionType	preSolveFunc: CollisionPre-	separateFunc: CollisionSepa-
typeB: CollisionType	SolveFunc	rateFunc
beginFunc: CollisionBegin-	postSolveFunc: Collision-	userData: DataPointer
Func	PostSolveFunc	

ContactBufferHeader:

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

ContactBuffer:

header: ContactBufferHeader
contacts: array(Contact)

8.4.2 Assumptions

spaceInit is called before any other access programs.

8.4.3 Semantics

spaceCollide Shapes:	Input:	spaceCollideShapes accepts two Shape pointers, a Collision ID and a Space pointer as inputs.
	Exceptions:	None.
	Transition:	spaceCollideShapes tests if the input Shapes can be collided using queryReject . If it fails, it returns the input ID. Otherwise, it performs collision detection and makes a new CollisionInfo structure. If a collision occurs, the function modifies the number of Contacts for the input Space, updates the Arbiter for the input Shapes, calls the Arbiter's collision handler functions and updates the Arbiter's timestamp. Otherwise, no further transitions are made. In either case, the function returns the ID of the generated CollisionInfo structure.
	Output:	spaceCollideShapes returns a CollisionID as output.
spaceStep:	Input:	spaceStep accepts a Space pointer and a as inputs.
	Exceptions:	None.

Transition: spaceStep updates the input Space following the specified timestep (input double). If the timestep is zero, the function exits immediately. Otherwise, it updates the Space's timestamp and current timestep, resets space lists and locks the Space. While the Space is locked, the function calculates new positions of Bodies in the Space and collides Shapes as necessary. before unlocking the Space without running post-step callbacks. Next, it locks the Space once again, clears cached Arbiters, pre-processes the Arbiters, updates the velocities of Bodies in the Space, applies cached impulses, runs the impulse solver, and then runs post-solve callbacks on the Arbiters. Finally, it unlocks the Space and runs post-step callbacks.

Output: None.

8.4.4 Local Functions

**spaceUse
WildcardDe-
faultHandler:**

Input: spaceUseWildcardDefaultHandler accepts a Space pointer as input.

Exceptions: None.

Transition: The function sets the Space to use wildcards and copies collisionHandlerDefault to the Space's default handler. Called by [spaceAddDefaultCollisionHandler](#).

Output: None.

**spaceAlloc
Contact-
Buffer:**

Input: spaceAllocContactBuffer accepts a Space pointer as input.

Exceptions: None.

Transition: spaceAllocContactBuffer heap-allocates a new contact buffer and adds it to the input Space's allocated buffers. Called by [spacePushFreshContactBuffer](#) to allocate a new ContactBufferHeader.

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

contactBufferHeaderInit:	Input:	contactBufferHeaderInit accepts a ContactBufferHeader pointer, a Timestamp and another ContactBufferHeader pointer as input.
	Exceptions:	None.
	Transition:	contactBufferHeaderInit initializes the first input ContactBufferHeader. It modifies its timestamp to the given Timestamp, its next header to be the next header of the second input ContactBufferHeader (or to the first input header if the second one is null), and its number of Contacts to zero. Called by spacePushFreshContactBuffer to initialize a ContactBufferHeader.
	Output:	contactBufferHeaderInit returns a pointer to the initialized ContactBufferHeader.

9 MIS of the Vector Module

9.1 Module Name: Vector

9.2 Uses

This module only uses standard libraries.

9.3 Syntax

9.3.1 Exported Constants

VECT_ERR, zeroVect: Vector

VECT_ERR := {INT_MAX, INT_MIN}

zeroVect := {0.0, 0.0}

PI:= \mathbb{R}

9.3.2 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	-

9.4 Semantics

9.4.1 State Variables

Vector:

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

9.4.2 Semantics

vect:	Input:	vect accepts two doubles as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vect returns a new Vector created from the input doubles.
vectEqual:	Input:	vectEqual accepts two Vectors as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectEqual compares the values of the input Vectors and returns true if they are equal, and false otherwise.
vectAdd:	Input:	vectAdd accepts two Vectors as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectAdd returns the sum of the input Vectors.
vectSub:	Input:	vectSub accepts two Vectors as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectSub returns the difference of the input Vectors.
vectMult:	Input:	vectMult accepts a Vector and a double as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectMult returns the scalar multiple of the input Vector with the input double.
vectNeg:	Input:	vectNeg accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.

	Output:	vectNeg returns the negative of the input Vector.
vectDot:	Input:	vectDot accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectDot returns the dot product of the input Vectors.
vectCross:	Input:	vectCross accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectCross calculates the cross product of the input Vectors and returns the z -component of the product as a double.
vectForAngle:	Input:	vectForAngle accepts a double as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectForAngle computes the Vector corresponding to the input angle (double), measured from the x -axis, and returns the result.
vectToAngle:	Input:	vectToAngle accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.
	Output:	vectToAngle calculates the angle between the input Vector and the x -axis and returns the result as a double.
vectRotate:	Input:	vectRotate accepts two Vectors as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	vectRotate rotates the first input Vector by the second using complex multiplication returns the resultant Vector as output.
vectLength:	Input:	Each vectLength function accepts a Vector as input.
	Exceptions:	None.
	Transition:	None.

Output: vectLength and vectLengthSq calculates the regular and squared length of the input Vector, respectively, and returns the result as a double.

10 MIS of the Collision Solver Module

10.1 Module Name: Collision

10.2 Uses

Rigid Body Module, Shape Module, Vector Module

10.3 Syntax

10.3.1 Exported Constants

N/A

10.3.2 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	-

10.4 Semantics

10.4.1 State Variables

SupportPoint:

p: Vector
index: CollisionID

MinkowskiPoint:

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

SupportContext:

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

EdgePoint:

p: Vector
hash: HashValue

Edge:

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

ClosestPoints:

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

10.4.2 Access Program Semantics

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

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

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

10.4.3 Local Constants

BuiltinCollisionFuncs: array(CollisionFunc)

BuiltinCollisionFuncs := {CircleToCircle, CollisionError, CollisionError, CircleToSegment, SegmentToSegment, CollisionError, CircleToPoly, SegmentToPoly, PolyToPoly}

CollisionFuncs := BuiltinCollisionFuncs

10.4.4 Local Functions

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

	Exceptions:	collisionInfoPushContact may throw a CollisionContactOverflow exception when the number of Contacts of the input CollisionInfo exceeds MAX_CONTACTS_PER_ARBITER.
	Transition:	collisionInfoPushContact pushes a new Contact structure into the input CollisionInfo's Contacts array with the other input parameters and updates its number of Contacts accordingly. Called by the ShapeToShape collision functions to add new contact points and by closestPoints in the collision functions for SegmentShapes and PolyShapes.
	Output:	None.
SupportPoint:	Input:	Each SupportPoint function accepts a Shape pointer of the Shape type corresponding to the function's prefix and a Vector as inputs.
	Exceptions:	None.
	Transition:	None.
	Output:	Each SupportPoint creates a new SupportPoint with the input Shape's transformed center (CircleShapes), endpoint (SegmentShape) or vertex (PolyShape), with the appropriate index of the point as its CollisionID. Each corresponding function is used by the appropriate ShapeToShape function in generating the SupportPointContext to be passed to GJK .
support:	Input:	support accepts a SupportContext pointer and a Vector as inputs.
	Exceptions:	None.
	Transition:	support calculates the maximal point on the Minkowski difference of two shapes along a particular axis. It generates two SupportPoints using the SupportPointFunc functions and Shapes contained in the input SupportContext and the input Vector, and creates a new MinkowskiPoint with these SupportPoints. Used in the calculations of GJK and EPA .
	Output:	support returns the new MinkowskiPoint as output.
supportEdgeFor:	Input:	Each supportEdgeFor function accepts a Shape pointer of the corresponding Shape type and a Vector as inputs.

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

	Transition:	closestPointsNew finds the closest edge to the origin (0, 0) on the Minkowski difference of two Shapes, which is obtained by using closestT and lerpT with the input MinkowskiPoints. This is used to calculate the closest points on the surface of two Shapes, as well as the distance and the minimum separating axis between them. The function then generates a new ClosestPoints structure using the calculated data and the concatenated IDs of the input MinkowskiPoints. Used to compute closest points in EPA and GJK .
	Output:	closestPointsNew returns the new ClosestPoints as output.
contactPoints:	Input:	contactPoints accepts two Edge structures, a ClosestPoints structure and a CollisionInfo pointer as inputs.
	Exceptions:	None.
	Transition:	contactPoints finds contact point pairs on the surfaces of the input support Edges and pushes a new Contact structure into the input CollisionInfo's Contacts array. This is used in ShapeToShape functions involving SegmentShapes and PolyShapes (except for CircleToPoly).
	Output:	None.
ShapeToShape:	Input:	Each ShapeToShape function accept two pointers to the corresponding Shape types and a CollisionInfo pointer as inputs.
	Exceptions:	None.
	Transition:	Each ShapeToShape function calls GJK to find the ClosestPoints for the two input Shapes and uses it to check if the current distance between the two Shapes is less than the minimum collision distance (usually determined by the sum of the Shapes' radii). If so, the function pushes a new Contact structure containing information about the Shapes' contact points into the Contacts array of the input CollisionInfo. These functions are stored in the exported CollisionFuncs array, and the appropriate function will be called by collide .
	Output:	None.

CollisionError:	Input:	CollisionError accepts two Shape pointers and a Collision-Info pointer as inputs.
	Exceptions:	CollisionError throws an eponymous exception when the types of the input Shapes are not in sorted order.
	Transition:	CollisionError throws an exception and aborts the program. This function is stored in the exported Collision-Funcs array and called by collide when the colliding Shape types are not in order.
	Output:	None.

[Please remove the commented out portions. They are distracting. Nothing will be lost, since it is all in version control. Moreover, I don't think we want to continue along the path of the previous documentation. It was too tied to the implementation. —SS]

[As mentioned above, we need a simpler design. One that is tied to the SRS, not the the existing code. The “easy” way of modifying the previous documentation was likely a mistake. It would be better if you thought about the problem following the design examples from class. —SS]

[You won't have time for completely redoing this document, but please indicate how you would change it in the revised documentation. —SS]

[This is actually the harder way to go i.e modifying previous documentation, I had to read through over a hundred pages, I thought the ideas here would be helpful in the design. Its not as easy to document the design of this library in the short amount of time during the term but I am working on revamping the entire documentation which might not be ready soon. —OO]