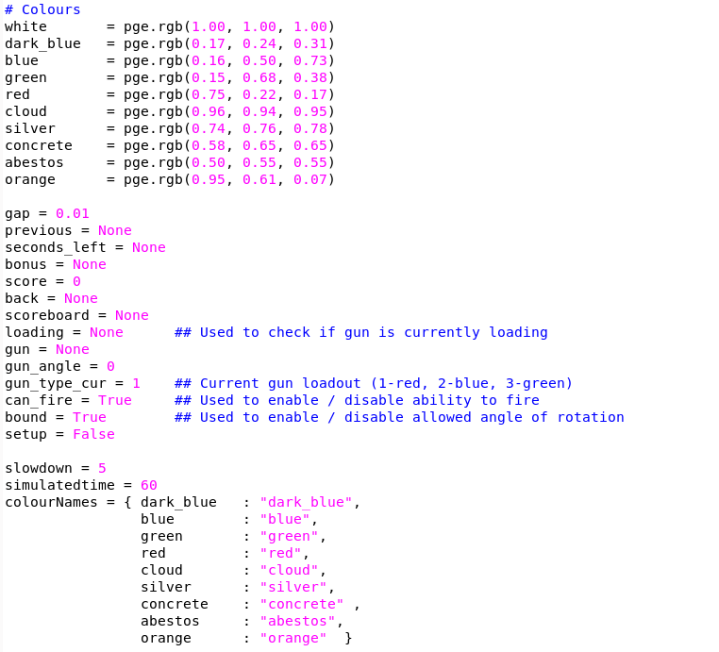
**Assessment Title and Tasks:**

**Integrating game engine and scripted code**

Changes made:

**Global variables added to 30005209.py**

Function .0 30005209.py

A series of variables were added to aid ease of function development. It also lowered a dependence on hard-coded variables, instead allowing ones to be changed at run-time.

Improvements: I would try to make sure all variables are coded here rather than hard-coded.

Graphical user interface, text

Description automatically generated**Displaying Score (30005209.py)**

Function .0 30005209.py

The function uses global variables to first check no scoreboard has been created, then creating one to display the score on the top layer (to avoid any form of collision).

Improvements: I would make display score use variables for both the background colour and the text to allow for changes to be made at run-time. Different levels could have different themes this way. I would make sure there is a defaulted function so that if no variables are given a ‘standard’ one is made.

**Displaying loading (30005209.py)**

Graphical user interface, text, application

Description automatically generatedThe function works very similarly to the display\_score() function but instead shows a text to denote it is loading.

Function .0 30005209.py

Improvements: Recognising how similar these are I would make both use a more general function. Including the improvements mentioned in the display\_score function I would also add a text variable to allow for each to be written, as well as a coordinate variable to denote where it should be placed. There is however, utility in making these more specialised as it stops functions from taking too many parameters – which could harm legibility.

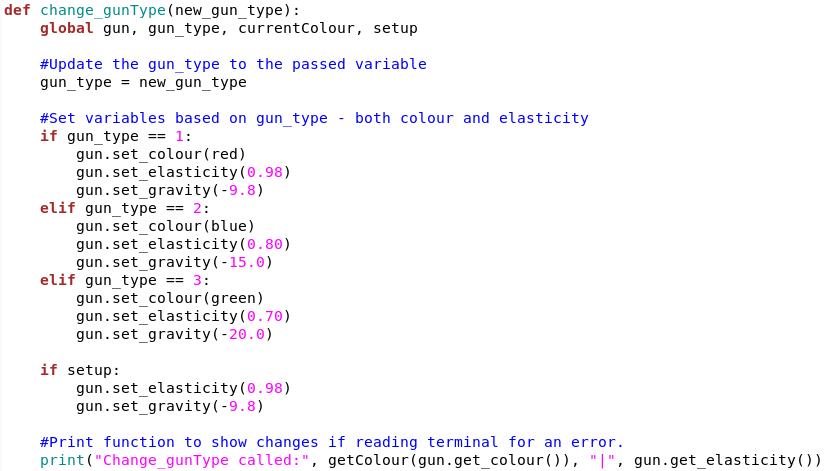
**Displaying fire (302005209.py)**

**Graphical user interface, text, application

Description automatically generated**This function also works the same as the above two. Further making arguments to creating a common function they all could share.

Function .0 30005209.py

Improvements: If I did decide to not make them all one function, I would definitely still combine both this and the display loading function. Instead making an else case after “if loading” to print out the fire command. The only benefit of them being separate functions (bar potential legibility) is if one did not want a colour to be passed as a variable and you wanted shoot to have a different colour to loading. I think this is unlikely and instead could also just account for that in the function.

**Changing Gun Information**

Graphical user interface, text, application

Description automatically generated

Function 5.1 pge.py

Function 5.0 30005209.py

****

Function 7 pgeif.i – done twice both python and C

Graphical user interface, application

Description automatically generated

Function 5.2 pgeif.c

Graphical user interface

Description automatically generated with medium confidence

Function 5.3 Pgeif.h

Graphical user interface, text, application, email

Description automatically generated

Function 5.4 twoDsim.c

Graphical user interface, text

Description automatically generated

Function 5.5 twoDsim.c

The code above starts within python making calls to arguments handled and converted into C and C objects. This is done both for the colour parameter – with arguments set\_colour, twoDsim\_set\_colour, etc. This is also done for elasticity and gravity– the code for this can be found in the appendix in the same files as the images above. With elasticity instead of colour in the function names – with doubles used instead of a deviceIF\_Colour. There is a setup Boolean to only enable the colours to have different elasticity/gravity after the gameboard is set. If this was not done, creating the level would also have to account for the effects variance of elasticity/gravity would have on each ball.

**Display Bonus**

Graphical user interface, text, application, email

Description automatically generated

Function 6.0 30005209.py

Displays text denoting a bonus has been scored after a period of time it is called again (which would remove the previously displayed text). This recursion sacrifices a bit of legibility for a simpler code-base with less functions. A “remove bonus” function would be excessive.

Improvements: Similar to above mentioned improvements a colour field and/or custom text variable would allow things that are not bonuses to pop up. Doing this again I would make display\_bonus a ‘display\_message’ function that has colours and text as input variables. I would then make ‘add\_bonus’ another function which not only calls display\_message (with appropriate given inputs) but also adds to the score.

**Deleting objects / Updating score**

**Graphical user interface, text, application

Description automatically generated**

Function 7.0 30005209.py

Graphical user interface, text, application, email, website

Description automatically generated

Function 7.1 30005209.py

This function deletes a passed pge object when provided the object and event. This is being used principally for the ‘chairs’ (later described) so an update to the score is also added.

Improvements: Further testament to decoupling the updating of the score method from the displaying bonus method is shown here. An improvement I would make is once again decouple these methods to allow for greater flexibility – changing behaviour via function calls not via editing of function code.

**Object creations (chair, board, net)**

Graphical user interface, text, application

Description automatically generated

Function 8.0 30005209.py

Each of these functions creates a simple shape that can be set at locations. This is to help encapsulate likely repeated behaviour (making boxes and the ‘chair’ shape).

Improvements: I would make them take colours as parameters as well as make both board and net take the height and width as variables for ease of customisation. An alternative would be to have ‘create\_object’ as a function with codes written that can be chosen within the function via if statements (recreating a switch statement).

**Create Level**

Text

Description automatically generated

Function 9.0 30005209.py

The create level function creates a series of rows of alternating circles automatically (this is called prior to the timer starting. It iterates through a list to get the coordinates.

**Increment Gun**

Graphical user interface, text

Description automatically generated

Function 10.0 30005209.py

The gun increments its value and then calls the change\_gun\_type based on the new value. This value is capped at 3 – due to that being the number of current settings.

Improvements: I would automate the number it’s being capped at rather than hardcoding it at 3. Either by making a global variable of the current number of settings or encapsulating that further in a struct. A further improvement would be to make the gun\_type\_cur use the modulus operator rather than the simplistic >3 it currently is using – that is personal preference as I am unsure of the speed benefits of either approach. I am assuming the modulus operator would be faster.

**Timer**

Text

Description automatically generated

Function 11.0 30005209

A picture containing text

Description automatically generated

Function 101.1 30005209.py

Text

Description automatically generated with low confidence

Function 11.2 30005209.py

The timer counts down – displaying the value in the corner, once that timer has run out it enacts the out\_of\_time function which closes the program after 4 seconds.

Improvements: I would again make the timer’s values of colour and position an inputted variable or global. I would also do the same for out\_of\_time. I think the decoupling is a good form as it allows extension of the code if needed.

**Spin Gun**

Graphical user interface, text, application

Description automatically generated

Function 12.0 30005209.py

The gun spins after being made no longer bound (this is used in another function to restrict it’s firing arc). It is then recursively called to keep spinning.

Improvements: I would make the function more abstract so objects can be passed rather than making this dedicated to the gun. A spinning gun function could be kept (for legibility) that would simply call the more abstract variant whilst passing the gun object. I would also like the abstract method (and the new spin gun method) to have an attached speed input variable to allow for greater flexibility.

**Placing the Gun**

**Graphical user interface, text, application

Description automatically generated**

Function 13.0 30005209.py

A gun with hard-coded variables is made.

Improvements: I would make it so the coordinates can be set via input variables this would allow the level to redefine where the gun could be made. One level where it on one side or the other rather than the defaulted centre. Also the guntype should be a variable so it can be initialised at a different starting value.

**Controls / Firing**

**Text

Description automatically generated**

Function 14.0 30005209.py

Text

Description automatically generated with medium confidence

Function 14.1 30005209.py

**Text

Description automatically generated**

Function 14.2 30005209.py

When the Up key is pressed the gun fires, provided it is not blocked (from firing recently). Pressing left or right alters its aim and firing adds a delay form when it can fire again. The reload function also updates the text at the top of the screen to denote if the player can fire again.

Improvements: One improvement I would like would be for the ‘bullet’ to originate at an arc from the ‘barrel’ as at the moment it simply moves from the centre at the angle the barrel is aiming. I also would like the starting location to be worked out rather than hard coded in to allow for ease of changing where the gun could start from.

**Testing**

**Text, application

Description automatically generated**

Function 16.0 30005209.py

The elastic\_test function (though it also tests the gravity), shoots three of one type of bullet before incrementing, this is then repeated for the other two types of bullets. This will show how the bullets are fired (speed, drag, etc) and then how they fall / bounce / collide when three of a kind is made.

Improvements: I would improve this test by automating the firing of all types of bullets (linking to a previous improvement made of automating the bullet types). As the code is currently written you can still rotate the gun (potentially leading to the three of a kind-connection not being made).

Graphical user interface

Description automatically generated with medium confidence Graphical user interface

Description automatically generated Graphical user interface

Description automatically generated

Testing results (Gravity): The images above show the falling difference between them as they fall (with different gravity. It is easier to see the difference whilst the code is running (images being harder to represent this).

Text

Description automatically generatedTesting results (Elasticity): When the net is placed down and each series of balls is allowed to fall uninterrupted by different ones you can see a marked difference in how much the energy is conserved between reactions with each other, chairs, and the walls. These inelastic collisions are also difficult to show via photos but are easily visible by adjusting the code as shown above. Careful note to make sure that the increment gun calls are still included should you wish to see the effects the blue/green values for elasticity.

Chart, bubble chart

Description automatically generatedA picture containing graphical user interface

Description automatically generated

Testing results (Create Level): Though originally the code had hard clean rows I found after adding the elasticity and the gravity a curved arc was made. I thought it provided greater challenge to the player having this pattern encourages the player to try to get the bullets to bounce off of the walls. The code on the left is the new pattern, whereas the code on the right shows the older pattern.

A picture containing square

Description automatically generated

Miscellaneous Testing: throughout the project I also performed many tests that were not formally bound within a function – such as the above in which I spawned in balls with different elasticity. They were spawned at the same height with the ‘net’ (as described earlier) along the bottom. This was to test to see how the inelastic collisions manifested at different variable values.

I have provided the files I believe to be relevant to the report below. If you would like to run this code or see any files not pasted below please clone the public repo for this program which can be found in all the headers of this document as well as below:

[www.github.com/30005209/gameEngineCw](http://www.github.com/30005209/gameEngineCw)

A copy of this report can also be found in the repository.

**examples/30005209/30005209.py**

#!/usr/bin/env python3

import pge, sys, math, random

from pygame.locals import \*

#

# bubble class which is used to record, colour, circle and children

# children are bubbles which connect below the current bubble

#

class bubble:

def \_\_init\_\_ (self, cir, col):

self.circle = cir

self.colour = col

self.circle.fix ()

self.circle.on\_collision (bubble\_hits\_bubble)

self.children = [] # no bubbles attached below this bubble

#

# addChild - adds bubble, b, as a child of self.

#

def addChild (self, b):

self.children += [b]

#

# removeChild - removes all bubbles in blist from the children field.

#

def removeChild (self, blist):

if self.children != []:

self.children = [x for x in self.children if x not in blist]

bubbleDict = {} # allows us to obtain a bubble list from a pge circle

currentCircle = None

currentColour = None

#

# printColour - print the colour string.

#

def printColour (c):

global colourNames

print(colourNames[c], end=' ')

def getColour(c):

global colourNames

return colourNames[c]

#

# dumpData - display all bubble colours and their chains and children

#

def dumpData (message):

print("dumpData", message)

print("bubbleDict", bubbleDict)

for c in list(bubbleDict.keys ()):

print("pge circle", end=' ')

print(" is in a list with: ", end=' ')

for l in bubbleDict[c]:

printColour (l.colour)

if l.children != []:

print(" [ children: ", end=' ')

for c in l.children:

printColour (c.colour)

print(", ", end=' ')

print("]", end=' ')

print(", ", end=' ')

print(" ")

#

# removeChildren - remove all bubbles in blist from all children in all bubbles.

#

def removeChildren (blist):

for circle in list(bubbleDict.keys ()):

for b in bubbleDict[circle]:

b.removeChild (blist)

#

# unfreezeList - unfreeze all bubbles in blist and return any children bubbles

# of these bubbles. Remove any reference to these circles/bubbles

# as they will be deleted as soon as they hit the bottom edge.

#

def unfreezeList (blist):

todo = []

for b in blist:

print("bubble", b, "should be unfixed")

b.circle.unfix ()

if b.circle in bubbleDict:

todo += bubbleDict[b.circle]

del bubbleDict[b.circle]

todo += b.children

removeChildren (todo)

return todo

#

# unfreezeAll - unfreeze all bubbles attached to circle (via children)

#

def unfreezeAll (circle):

global bubbleDict

print("unfreezeAll", circle)

dumpData ("unfreezeAll")

blist = bubbleDict[circle]

while blist != []:

blist = unfreezeList (blist)

currentCircle.unfix ()

dumpData ("finished unfreeze")

#

# bubble\_hits\_bar - the callback for the bubble hitting the top edge.

#

def bubble\_hits\_bar (o, e):

global bubbleDict, currentCircle

print("bubble\_hits\_bar", currentCircle)

if currentCircle != None:

if currentCircle in bubbleDict:

print("odd the bubble is already registered")

else:

initChain (currentCircle, currentColour)

currentCircle = None

dumpData ("end of bubble\_hits\_bar")

#

# initChain - create an initial bubble in a chain of one.

#

def initChain (circle, colour):

global bubbleDict

b = bubble (circle, colour)

bubbleDict[circle] = [b] # the first bubble in a chain is keyed via a circle

#

# updateChains - adds currentCircle/currentColour bubble to the chain specified by circle.

#

def updateChains (circle):

global bubbleDict

print("updateChains")

blist = bubbleDict[circle]

print("blist", blist, "len (blist) =", len (blist))

blist += [bubble (currentCircle, currentColour)]

print("blistis now", blist, "len (blist) =", len (blist))

# and update all circle -> chain entries

for b in blist:

bubbleDict[b.circle] = blist

#

# addBubble - Prerequiste : circle is a pge object which has not been deleted.

# Postrequisite: assuming that circle is already known in the bubbleDict

# and if the collided chain has two bubbles of the same colour

# then unfreeze these bubble and their children.

# A same colour collision is appended to the chain.

# A different colour collision creates a new chain with

# a single bubble.

#

def addBubble (circle):

global bubbleDict

print("addBubble")

if circle in bubbleDict:

blist = bubbleDict[circle]

print(blist, len (blist))

if blist[0].colour == currentColour:

print("same colour", blist)

if len (blist) == 2:

unfreezeAll (circle)

return

else:

print("not enough of the same colour bubbles to unfreeze them")

updateChains (circle)

else:

print("bubble hit is a different colour")

b = bubble (currentCircle, currentColour)

blist[-1].addChild (b)

bubbleDict[currentCircle] = [b]

dumpData ("end of addBubble")

else:

print("addBubble - does not know about circle", circle)

#

# bubble\_hits\_bubble - call back for a circle hitting a frozen bubble.

#

def bubble\_hits\_bubble (o, e):

global currentCircle

print("bubble hits bubble, currentCircle =", currentCircle)

if currentCircle != None:

b = e.collision\_between ()

for o in b:

if o.is\_fixed ():

print("object, o, is fixed", o)

else:

print("object, o, is not fixed", o)

if not currentCircle.is\_fixed ():

addBubble (o)

currentCircle = None

dumpData ("end of bubble\_hits\_bubble")

print("starting frozenbubble")

pge.interactive ()

# pge.batch ()

pge.record ()

t = pge.rgb (1.0/2.0, 2.0/3.0, 3.0/4.0)

#wood\_light = pge.rgb (166.0/256.0, 124.0/256.0, 54.0/256.0)

#wood\_dark = pge.rgb (76.0/256.0, 47.0/256.0, 0.0)

#red = pge.rgb (0.65, 0.1, 0.2)

#steel = pge.rgb (0.5, 0.5, 0.5)

#copper = pge.rgb (0.5, 0.3, 0.2)

#gold = pge.rgb (0.8, 0.6, 0.15)

ball\_size = 0.02

boarder = 0.01

#green = pge.rgb (0.1, 0.6, 0.2)

#blue = pge.rgb (0.0, 0.0, 1.0)

#blue = pge.rgb (0.0, 100.0/255.0, 1.0)

# Colours

white = pge.rgb(1.00, 1.00, 1.00)

dark\_blue = pge.rgb(0.17, 0.24, 0.31)

blue = pge.rgb(0.16, 0.50, 0.73)

green = pge.rgb(0.15, 0.68, 0.38)

red = pge.rgb(0.75, 0.22, 0.17)

cloud = pge.rgb(0.96, 0.94, 0.95)

silver = pge.rgb(0.74, 0.76, 0.78)

concrete = pge.rgb(0.58, 0.65, 0.65)

abestos = pge.rgb(0.50, 0.55, 0.55)

orange = pge.rgb(0.95, 0.61, 0.07)

gap = 0.01

previous = None

seconds\_left = None

bonus = None

score = 0

back = None

scoreboard = None

loading = None ## Used to check if gun is currently loading

gun = None

gun\_angle = 0

gun\_type\_cur = 1 ## Current gun loadout (1-red, 2-blue, 3-green)

can\_fire = True ## Used to enable / disable ability to fire

bound = True ## Used to enable / disable allowed angle of rotation

setup = False

slowdown = 5

simulatedtime = 60

colourNames = { dark\_blue : "dark\_blue",

blue : "blue",

green : "green",

red : "red",

cloud : "cloud",

silver : "silver",

concrete : "concrete" ,

abestos : "abestos",

orange : "orange" }

def finish\_game (event, param):

#Close the program

sys.exit (0)

def placeBoarders (thickness, color):

print("placeBoarders")

n = pge.box (0.0, 1.0-thickness-0.1, 1.0, thickness, color).fix ()

e = pge.box (1.0-thickness, 0.0, thickness, 1.0, color).fix ()

s = pge.box (0.0, 0.0, 1.0, thickness, color).fix ()

w = pge.box (0.0, 0.0, thickness, 1.0, color).fix ()

return n, e, s, w

def fire (e, colour):

global currentColour, currentCircle, gun\_angle

mouse = pge.pyg\_to\_unit\_coord (e.pos)

currentColour = colour

currentCircle = pge.circle (0.5, 0.21, 0.03, currentColour).mass (1.0)

currentCircle.put\_yvel (7\*mouse[1])

currentCircle.put\_xvel (math.sin(gun\_angle/180.0)\*-6.0)

def fire\_circle (e):

if e.button == 1:

fire (e, red)

elif e.button == 2:

fire (e, blue)

elif e.button == 3:

fire (e, green)

def mouse\_press (e):

#global currentCircle, currentColour

print(pge.pyg\_to\_unit\_coord (e.pos))

#if currentCircle == None:

# fire\_circle (e)

def myquit (e):

print("goodbye, dumping world")

pge.dump\_world ()

sys.exit (0)

def delete\_ball (o, e):

global score

print("delete\_ball called")

p = e.collision\_between ()

if p != None and p != []:

for i in p:

if i != o:

update\_score(10)

i.rm ()

def fire\_bubble\_key(e = None, f = None):

global currentCircle, gun, currentColour, seconds\_left

#Provided the game isn't over set the variables from gun

if currentCircle == None and seconds\_left > 0:

currentColour = gun.get\_colour()

currentCircle = pge.circle (0.5, 0.21, 0.03, currentColour)

currentCircle.mass (1.0)

currentCircle.put\_yvel(8.5)

#Notably the elasticity...

currentCircle.set\_elasticity(gun.get\_elasticity())

#...and the Gravity on a per obj basis

currentCircle.set\_gravity(gun.get\_gravity())

#Print some info to terminal

print("GRAVITY: ", currentCircle.get\_gravity())

print(gun.get\_elasticity())

scale = (gun\_angle - 180.0) / 180.0

#print("scale = ", scale)

scale \*= -20.0

#print ("vec =[", scale, ", 7.0]")

currentCircle.put\_xvel(scale)

update\_score(-5)

def fire\_bubble ():

global currentCircle, currentColour, score

if currentCircle == None:

currentColour = red

currentCircle = pge.circle (0.5, 0.1, 0.03, currentColour)

currentCircle.mass (1.0)

currentCircle.put\_yvel (8.0)

# pge.dump\_world ()S

def reload(e = None, f = None):

global gun, can\_fire

#Update ability to fire

display\_fire()

can\_fire = True

def increment\_gun(e = None, f = None):

global gun\_type\_cur

# Increment the current gun to be another setting

gun\_type\_cur += 1

#Should it exceed the current max of settings - loop to 0

if gun\_type\_cur > 3:

gun\_type\_cur = 1

#Ouput change in terminal

print("increment\_gun called: ", gun\_type\_cur)

#Change gun to new setting

change\_gunType(gun\_type\_cur)

def key\_pressed (e):

global gun, gun\_angle, gun\_types, gun\_type\_cur, can\_fire, bound

temp\_a = gun\_angle

if e.key == K\_ESCAPE:

myquit (e)

#Press down to change the ammo/gun type

elif e.key == K\_DOWN:

increment\_gun()

#Press up to fire

elif e.key == K\_UP:

#If allowed to fire

if can\_fire:

fire\_bubble\_key()

display\_loading()

can\_fire = False

pge.at\_time(2.0, reload)

#Press right to turn one way

elif e.key == K\_RIGHT:

gun\_angle += 5

#Press left to turn the other way

elif e.key == K\_LEFT:

gun\_angle -= 5

#So long as the gun is bound keep it between 100 and 260

if bound:

if gun\_angle < 100:

gun\_angle = 100

elif gun\_angle > 260:

gun\_angle = 260

if temp\_a != gun\_angle:

gun\_angle %=360

gun.rotate (gun\_angle \* math.pi / 180.0)

#print(gun\_angle)

def place\_bubble (x, colour):

global currentColour, currentCircle, gun

#Change gun type based on colour provided

if colour == red:

change\_gunType(1)

elif colour == blue:

change\_gunType(2)

elif colour == green:

change\_gunType(3)

# Set local variable to information from gun

currentColour = gun.get\_colour()

currentCircle = pge.circle (0.5, 0.21, 0.03, currentColour).mass (1.0)

currentCircle.put\_yvel(8.5)

currentCircle.put\_xvel ((0.5-x)\*-6.0)

pge.run (0.2)

def change\_gunType(new\_gun\_type):

global gun, gun\_type, currentColour, setup

#Update the gun\_type to the passed variable

gun\_type = new\_gun\_type

#Set variables based on gun\_type - both colour and elasticity

if gun\_type == 1:

gun.set\_colour(red)

gun.set\_elasticity(0.98)

gun.set\_gravity(-9.8)

elif gun\_type == 2:

gun.set\_colour(blue)

gun.set\_elasticity(0.80)

gun.set\_gravity(-15.0)

elif gun\_type == 3:

gun.set\_colour(green)

gun.set\_elasticity(0.70)

gun.set\_gravity(-20.0)

if setup:

gun.set\_elasticity(0.98)

gun.set\_gravity(-9.8)

#Print function to show changes if reading terminal for an error.

print("Change\_gunType called:", getColour(gun.get\_colour()), "|", gun.get\_elasticity())

#

#

#

def display\_bonus (e = None, f = None):

global bonus

# Check there is no bonus currently in play (delete if there is)

if bonus is None:

# Display text denoting a bonus has been scored

bonus = pge.text (0.3, 0.3, "bonus score", white, 100, -1)

pge.at\_time(1.0, display\_bonus)

else:

bonus.rm()

bonus = None

def delete\_me(pge\_obj, pge\_event):

global score, bonus

#Removes the object from the scene

pge\_obj.rm()

#Adds score

update\_score(2)

#display\_bonus()

def chair(x,y,width,height, is\_left):

thickness = 0.01

#Create the object out of the back, seat and leg - facing the correct way

if is\_left:

back = pge.box(x + width - thickness, y, thickness, height, cloud).fix()

seat = pge.box(x, y+height/2 -thickness/2, width, thickness, cloud).fix()

leg = pge.box(x, y, thickness, height/2 - thickness/2, cloud).fix()

else:

back = pge.box(x,y, thickness, height, cloud).fix()

seat = pge.box(x, y + height / 2, width, thickness, cloud).fix()

leg = pge.box(x + width - thickness, y, thickness, height/2, cloud).fix()

#Define what happens on collisions

back.on\_collision (delete\_me)

seat.on\_collision (delete\_me)

leg.on\_collision (delete\_me)

def board(x,y):

#Create a horizontal box for level creation

width, height = 0.6, 0.05

thickness = 0.05

pge.box(x,y, width, height, orange).fix()

def net(x,y):

#Create a thinner than board horizontal box for level creation (and testing)

width, height = 1.0, 0.005

thickness = 0.05

pge.box(x,y, width, height, white).fix()

def display\_score():

global score, back, scoreboard

#Create a blue box to act as the backing for a scoreboard

back = pge.box(0.01, 0.9, 0.98, 1.0, blue).fix()

#If a scoreboard has been made delete it

if scoreboard != None:

scoreboard.rm()

#Create scoreboard containing the score written as white text on the top layer

scoreboard = pge.text(0.01, 1.0, "Score:" + str(score), white, 100, 1)

def display\_loading():

global score, back, loading

#Create a blue box to act as the backing for a scoreboard

back = pge.box(0.01, 0.9, 0.98, 1.0, blue).fix()

#If a loading sign has been made delete it

if loading != None:

loading.rm()

#Create loading sign containing the score written as white text on the top layer

loading = pge.text(0.65, 1.0, "Loading", white, 100, 1)

def display\_fire():

global score, back, loading

#Create a blue box to act as the backing for a scoreboard

back = pge.box(0.01, 0.9, 0.98, 1.0, blue).fix()

#If a loading sign has been made delete it

if loading != None:

loading.rm()

#Create loading sign containing the score written as white text on the top layer

loading = pge.text(0.65, 1.0, "Shoot!!", white, 100, 1)

def randomise\_gun():

#Calls the chang gun function with a randomised value

change\_gunType(random.randrange(1,4))

def update\_score(i):

global score

#Add the given variable to the score

score += i

#Redisplay the new score

display\_score()

def createLevel (e = None, f = None):

global gun, setup

# Create an alternating list of bubbles across the screen across three rows

# Increments are done so that colours dont fall upon making three of them.

setup = True

for x in [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.805]:

place\_bubble (x, gun.get\_colour())

increment\_gun()

for x in [0.16, 0.27, 0.38, 0.49, 0.60, 0.72]:

place\_bubble (x+0.07, gun.get\_colour())

increment\_gun()

#Extra increments were made as without them created a 3 line combination

increment\_gun()

increment\_gun()

for x in [0.25, 0.38, 0.50, 0.62, 0.75]:

place\_bubble (x, gun.get\_colour())

increment\_gun()

setup = False

increment\_gun()

def timer (e = None, f = None):

global seconds\_left, previous

#Check seconds\_left is higher than 0

if seconds\_left >= 0:

#recursivly call this function

pge.at\_time (1.0, timer)

s = "%d" % seconds\_left

#destroy the previous implementation of 'previous'

if previous != None:

previous.rm ()

#Set previous to new text value and decrement seconds\_left

previous = pge.text (0.8, 0.9, s, white, 100, 1)

seconds\_left -= 1

#Once seconds\_left reaches zero call 'out\_of\_time

if seconds\_left == 0:

out\_of\_time()

def spin\_gun():

global bound

#Make it so the gun is no longer restricted in its turning arc

bound = False

#Start spinning

spin\_callback()

def spin\_callback(e = None, f = None):

global gun, gun\_angle, bound

#Increment the angle of the gun

gun\_angle +=1

#Output to the terminal to show changes

print("called spin\_callback, angle=", gun\_angle)

#Call this again recursivley

pge.at\_time (4.0 / 360.0, spin\_callback)

#Cap it using the modulus operator to stop the value getting too high

# and so harder to read in the terminal

gun\_angle %=360

#Rotate the gun based on it's new value relative to pi/180

gun.rotate(gun\_angle \* math.pi / 180.0)

def out\_of\_time ():

global score

#Ouptut text showing the score

pge.text (0.3, 0.7, "Score: " + str(score), white, 100, 1)

#After 4 seconds finish the game

pge.at\_time (4.0, finish\_game)

def place\_gun():

global gun, gun\_angle, gun\_type

thickness = 0.02

height = 0.1

gun\_angle = 180

#Create a box with the given variables

gun = pge.box(0.5, 0.03, thickness, height, cloud).fix()

#Set the defaulted gun type to 1 (red)

change\_gunType(1)

def elastic\_test(e = None, f = None):

global gun

# Place a net in the way to help further should the elasticity

net(0.01, 0.01)

#Fire three of one type then increment

pge.at\_time(0.5,fire\_bubble\_key)

pge.at\_time(1.5,fire\_bubble\_key)

pge.at\_time(2.5,fire\_bubble\_key)

pge.at\_time(2.6, increment\_gun)

#Fire three of one type then increment

#pge.at\_time(4.5,fire\_bubble\_key)

#pge.at\_time(5.5,fire\_bubble\_key)

#pge.at\_time(6.5,fire\_bubble\_key)

#pge.at\_time(6.6, increment\_gun)

#Fire three of one type then increment

#pge.at\_time(10.5,fire\_bubble\_key)

#pge.at\_time(12.5,fire\_bubble\_key)

#pge.at\_time(14.5,fire\_bubble\_key)

def make\_chairs():

chair(0.1,0.1,0.05, 0.05, True)

chair(0.2,0.2,0.05, 0.05, False)

chair(0.3,0.1,0.05, 0.05, True)

chair(0.7,0.1,0.05, 0.05, True)

chair(0.8,0.2,0.05, 0.05, False)

chair(0.9,0.1,0.05, 0.05, True)

def main ():

global g, concrete, slowdown, seconds\_left, gun

n, e, s, w = placeBoarders (boarder, concrete)

s.on\_collision (delete\_ball)

n.on\_collision (bubble\_hits\_bar)

print("before run")

pge.gravity ()

pge.dump\_world ()

pge.draw\_collision (False, False)

pge.slow\_down (6.0) # slows down real time by a factor of

pge.fps (200)

pge.display\_set\_mode ([800, 800])

pge.register\_handler (mouse\_press, [MOUSEBUTTONDOWN])

pge.register\_handler (myquit, [QUIT])

pge.register\_handler (key\_pressed, [KEYDOWN])

display\_score()

display\_fire()

place\_gun()

make\_chairs()

#Decomment/comment to show testing for elastic as gravity test

#elastic\_test()

#Decomment/comment to create level - make sure other tests are commented

createLevel()

#Decomment/comment to spin gun - make sure other tests/level creation

#are commented

#spin\_gun()

seconds\_left = 10 \* slowdown

timer ()

pge.run (simulatedtime)

pge.run (10.0)

pge.finish\_record ()

print("before main()")

main ()

**python/pge.py**

#!/usr/bin/env python3

import pgeif

import pygame

import sys

import struct

import time

import math

from pygame.locals import \*

colour\_t, box\_t, circle\_t, spring\_t, fb\_box\_t, fb\_circle\_t, fb\_text\_t = list(range(7))

id2ob = {}

ob2id = {}

batch\_d, pyg\_d = list(range(2))

device = None

opened = False

output = None

lastDelay = 0.0

debugging = False

foreground= []

background= []

colours = []

levels = {}

resolution = None

fullscreen = False

screen\_initialised = False

pyevent2func = {}

call = {}

idTOcol = {}

nextFrame = 1

screen = None

backcanvas = None

program\_name = 'pge'

version\_number = '4.0'

Black = (0, 0, 0)

framesPerSecond = 100.0

frame\_no = 0

slow\_down\_factor = 1.0

allowed\_events = [USEREVENT+1]

id2func = {}

idcount = 0

\_record = False

font\_size = None

font = None

fps\_text = None

last\_fps = 0

#

# \_printf - keeps C programmers happy :-)

#

def \_printf (format, \*args):

print(str (format) % args, end=' ')

#

# \_debugf - \_printf when debugging is True

#

def \_debugf (format, \*args):

global debugging

if debugging:

print(str (format) % args, end=' ')

#

# \_errorf - generate an error associated with pge and raise an exception.

#

def \_errorf (format, \*args):

m = str (format) % args

sys.stdout.write ("pge: " + m)

raise Exception (m)

sys.exit (1)

#

# \_internalf - generate an internal error and raise an exception.

#

def \_internalf (format, \*args):

m = str (format) % args

sys.stdout.write ("internal error in pge: " + m)

raise Exception (m)

sys.exit (1)

class \_myfile:

def \_\_init\_\_ (self, c):

self.contents = c

self.length = len (self.contents)

self.pos = 0

\_debugf ("length %d bytes", self.length)

def read (self, n):

# print "read called for", n, "bytes", self.length-self.pos, "available"

if self.pos + n <= self.length:

b = self.contents[self.pos:self.pos+n]

self.pos += n

return b

else:

\_printf ("unexpected eof reached in frame %d\n", nextFrame)

sys.exit (1)

def close (self):

pass

def left (self):

return self.length-self.pos

def \_emit\_short (s):

global output

output.write (struct.pack ('!H', s))

def \_emit\_card (c):

global output

output.write (struct.pack ('!I', c))

def \_emit\_double (d):

global output

d = (float) (d)

output.write (struct.pack ('d', d))

def \_emit\_fract (f):

global output

if f == 0:

output.write (struct.pack ('B', 0))

elif f == 1:

output.write (struct.pack ('B', 1))

elif f < 1.0:

output.write (struct.pack ('B', 2))

output.write (struct.pack ('!QQ', f\*10000.0, 10000.0))

else:

w = int (f)

f -= w

output.write (struct.pack ('B', 3))

output.write (struct.pack ('!QQQ', w, f\*10000.0, 10000.0))

#

# object - the class of objects in the 2D world. An object can either

# be a colour, circle, polygon, spring or text. A text

# object can not exist in level 0. Objects are drawn

# from lowest level to highest level. Only objects in

# level 0 are processed by the physics engine.

# All other levels are eye candy.

#

class object:

def \_\_init\_\_ (self, t, o, c = None, level = None):

self.deleted = False

self.type = t

self.o = o

self.fixed = False

self.param = None

self.kg = None

self.collisionWith = []

self.collisionp = None

self.springp = None

self.w = None

if c == None:

self.c = self

else:

self.c = c

self.level = level

def \_id (self):

return self.o

def \_get\_3\_colour (self):

global idTOcol

self.\_check\_colour ()

i = pgeif.h2l (self.\_get\_pgeif\_colour ())

if i in idTOcol:

return idTOcol[i]

\_internalf ("3 colour triple should have been defined")

def \_get\_pgeif\_colour (self):

self.\_check\_colour ()

return self.o[-1]

def \_draw (self):

global screen, device

if self.type == fb\_box\_t:

if device == pyg\_d:

c = self.get\_colour ().\_get\_3\_colour ()

# print c

x = (int) (self.o[0] \* resolution[0])

y = (int) (self.o[1] \* resolution[1])

w = (int) ((self.o[2] - self.o[0]) \* resolution[0])

h = (int) ((self.o[5] - self.o[1]) \* resolution[1])

pygame.draw.rect (screen, c, (x, flip (y), w, h))

else:

self.\_emit\_fill\_polygon ()

elif self.type == fb\_circle\_t:

if device == pyg\_d:

c = self.get\_colour ().\_get\_3\_colour ()

# print c

x = (int) (self.o[0] \* resolution[0])

y = (int) (self.o[1] \* resolution[1])

r = (int) (self.o[2] \* resolution[0])

pygame.draw.circle (screen, c, (x, flip (y)), r, 0)

else:

self.\_emit\_fill\_circle ()

elif self.type == colour\_t:

pass

elif self.type == fb\_text\_t:

if device == pyg\_d:

c = self.get\_colour ().\_get\_3\_colour ()

font = get\_font (self.o[3])

t = font.render (self.o[2], True, c)

textpos = t.get\_rect ()

textpos.top = screen.get\_rect ().bottom - (int) (resolution[1] \* self.o[1])

textpos.left = screen.get\_rect ().left + (int) (resolution[0] \* self.o[0])

screen.blit (t, textpos)

def \_emit\_fill\_circle (self):

output.write (struct.pack ("3s", "dC"))

\_emit\_fract (self.o [0]) # x pos

\_emit\_fract (self.o [1]) # y pos

\_emit\_fract (self.o [2]) # radius

\_emit\_short (self.o [3]) # colour

# print "\_emit\_fill\_circle, colour is ", self.o [3], self.o [0], self.o [1], self.o [2]

def \_emit\_fill\_polygon (self):

output.write (struct.pack ("3s", "dP"))

n = int ((len (self.o)-1)/2)

\_emit\_short (n)

ier = iter (self.o[:-1])

# print self.o

for x in ier:

\_emit\_fract (x)

\_emit\_fract (next (ier))

\_emit\_short (self.o [-1])

def \_name (self):

if self.type == colour\_t:

return "colour"

elif self.type == box\_t:

return "box"

elif self.type == circle\_t:

return "circle"

else:

\_printf ("fatal error, object not recognised\n")

sys.exit (1)

#

# velocity - Pre-condition: a circle or polygon object

# which is not fixed and exists at level 0.

# Post-condition: assign the velocity (vx, vy)

# to this object.

#

def velocity (self, vx, vy):

self.\_check\_type ([box\_t, circle\_t], "assign a velocity to a")

# self.\_check\_not\_fixed ("assign a velocity")

self.\_check\_not\_deleted ("a velocity")

# print "velocity for object", self.o, vx, vy

self.o = self.\_check\_same (pgeif.velocity (self.o, vx, vy))

return self

#

# accel - Pre-condition: a circle or polygon object

# which is not fixed and exists at level 0.

# Post-condition: assign the accel (vx, vy)

# to this object.

#

def accel (self, ax, ay):

self.\_check\_type ([box\_t, circle\_t], "assign an acceleration to a")

# self.\_check\_not\_fixed ("assign an acceleration")

self.\_check\_not\_deleted ("an acceleration")

self.o = self.\_check\_same (pgeif.accel (self.o, ax, ay))

return self

#

# fix - Pre-condition: the object is either a circle or polygon

# which exists in level 0.

# Post-condition: mark this object as fixed to its initial

# position.

#

def fix (self):

self.\_check\_type ([box\_t, circle\_t], "fix a")

self.\_check\_not\_deleted (" a fixed position")

# self.\_check\_no\_mass ("cannot fix " + self.\_name () + " as it has a mass")

self.fixed = True

self.o = self.\_check\_same (pgeif.fix (self.o))

# print "fix", self.o

# print "returning from fix"

return self

#

# unfix - Pre-condition: the object is either a circle or polygon

# which exists in level 0.

# Post-condition: mark this object as unfixed.

#

def unfix (self):

self.\_check\_type ([box\_t, circle\_t], "unfix a")

self.\_check\_not\_deleted (" a fixed position")

self.fixed = False

self.o = self.\_check\_same (pgeif.unfix (self.o))

print("returning from unfix")

return self

#

# is\_fixed - returns True if the object is fixed in the world.

#

def is\_fixed (self):

self.\_check\_type ([box\_t, circle\_t], "an is\_fixed test")

self.\_check\_not\_deleted (" an is\_fixed test")

if pgeif.is\_fixed (self.o):

self.\_assert (self.fixed, "python api cached a different result from the game engine")

return True

else:

self.\_assert (not self.fixed, "python api cached a different result from the game engine")

return False

#

# mass - Pre-condition: the object is either a circle or polygon

# which exists in level 0. The object must not be fixed.

# Post-condition: object is given mass, m.

#

def mass (self, m):

self.\_check\_type ([box\_t, circle\_t], "assign a mass to a")

# self.\_check\_not\_fixed ("assign a mass")

self.\_check\_not\_deleted (" a mass")

if m is None:

\_errorf ("cannot give value None as the mass\n")

self.kg = m

self.o = self.\_check\_same (pgeif.mass (self.o, m))

# print "mass", self.o

return self

#

# apply\_impulse - Pre-condition: the object is either a circle or polygon

# which exists in level 0. The object must not be fixed.

# Post-condition: the object is given an impulse along vector,

# unit\_vec, with a, magnitude.

# unit\_vec is a list of two elements [x, y]

# magnitude is a float.

#

def apply\_impulse (self, unit\_vec, magnitude):

self.\_check\_type ([box\_t, circle\_t], "assign an impulse to a")

# self.\_check\_not\_fixed ("assign an impulse")

self.\_check\_not\_deleted (" an impulse")

if (magnitude is None) or (unit\_vec is None):

return

pgeif.apply\_impulse (self.o, unit\_vec[0], unit\_vec[1], magnitude)

return self

#

# on\_collision\_with - Pre-condition: the object is either a circle or polygon

# which exists in level 0.

# Post-condition: when this object collides

# with any object in the list, another,

# then function, p, is called.

# The parameter, another, is a list of objects.

# p must be a function which receives two parameters

# an object and an event.

#

def on\_collision\_with (self, another, p):

if debugging:

print("ok registering call back", p, another)

self.collisionp = p

self.collisionWith = another

return self

#

# on\_collision - Pre-condition: the object is either a circle or polygon

# which exists in level 0.

# Post-condition: when this object collides call function, p.

# p is a function with two parameters, o and e.

# o is this object.

# e is the event.

#

def on\_collision (self, p):

self.on\_collision\_with ([], p)

return self

def \_check\_type (self, legal, message):

if not self.type in legal:

\_errorf ("you cannot %s %s object\n", message, self.\_type\_name ())

def \_check\_not\_fixed (self, message):

if self.fixed:

\_errorf ("object %s is fixed, you cannot %s\n", self.\_type\_name (), message)

def \_check\_colour (self):

if self.type != colour\_t:

\_errorf ("object is expected to be a colour\n")

def \_param\_colour (self, message):

if self.type != colour\_t:

\_errorf (message)

def \_assert (self, b, message):

if not b:

\_errorf (message)

#

# rm - Pre-condition: the object can be a circle, polygon or text at any level.

# Post-condition: the object is removed.

#

def rm (self):

if not self.deleted:

# \_printf ("calling pgeif.rm\n")

if self.level == 0:

self.o = pgeif.rm (self.o)

else:

\_sub (self, self.level)

self.deleted = True

# \_printf ("returned from pgeif.rm\n")

def \_check\_not\_deleted (self, message):

if self.deleted:

\_errorf ("object has been deleted and now it is being given " + message)

def \_check\_no\_mass (self, message):

if self.kg != None:

\_errorf (message + "\n")

def \_check\_same (self, o):

if o == self.o:

return o

\_errorf ("internal error, object %d does not match self.o = %d\n", o, self.o)

def \_collision (self, between, e):

if debugging:

print("collision seen, between:", between)

if self.collisionWith == []:

if self.collisionp != None:

if debugging:

print("before collisionp")

self.collisionp (self, e)

if debugging:

print("after collisionp")

else:

found = False

for c in self.collisionWith:

for b in between:

if c == b:

found = True

if found and (self.collisionp != None):

self.collisionp (self, e)

#

# get\_param - Pre-condition: self is a polygon or circle object.

# Post-condition: returns user data given to this object.

#

def get\_param (self):

return self.param

#

# set\_param - Pre-condition: self is a polygon or circle object.

# Post-condition: assigns value as the user data to this object.

#

def set\_param (self, value):

self.param = value

return self

#

# set\_width - Pre-condition: self is a polygon or circle object.

# Post-confition: assigns an overall width value to an object.

# This value is not used by the physics engine.

# It is only present for user convenience.

#

def set\_width (self, value):

self.w = value

#

# get\_width - Pre-condition: self is a polygon or circle object which has

# had a user defined width set. Post-condition: returns the

# overall width value for this object.

#

def get\_width (self):

self.\_check\_type ([box\_t, circle\_t], "get the width")

return self.w

#

# get\_mass - Pre-condition: self is a polygon or circle object.

# Post-condition: return the mass of this object.

#

def get\_mass (self):

return self.kg

#

# get\_colour - Pre-condition: self is a polygon or circle object.

# Post-condition: return the colour of this object.

#

def get\_colour (self):

return self.c

#

# set\_colour - Pre-condition: self is a polygon or circle object.

# Post-condition: this object will be displayed in colour, c.

#

def set\_colour (self, c):

self.\_check\_type ([box\_t, circle\_t, fb\_box\_t, fb\_circle\_t], "set\_colour")

#Check the passed parameter is a colour

c.\_param\_colour ("first parameter to set\_colour is expected to be a colour")

#Set the colour

if self.type in [box\_t, circle\_t]:

pgeif.set\_colour (self.o, c.\_get\_pgeif\_colour ())

else:

self.o = self.o [:-1]

self.o += [c.\_get\_pgeif\_colour ()]

self.c = c

return self

#

# set\_elasticity- Pre-condition: self is a polygon or circle object.

# Post-condition: this object will be displayed in elasticity, c.

#

def set\_elasticity (self, elasticity):

self.\_check\_type ([box\_t, circle\_t], "change the elasticity for the object")

print("pge: ", elasticity)

return pgeif.set\_elasticity (self.o, elasticity)

#

# get\_elasticity- Pre-condition: self is a polygon or circle object.

# Post-condition: this object will change its elasticity.

#

def get\_elasticity (self):

self.\_check\_type ([box\_t, circle\_t, fb\_box\_t, fb\_circle\_t], "get elasticity")

return pgeif.get\_elasticity(self.o)

#

# set\_gravity- Pre-condition: self is a polygon or circle object.

# Post-condition: this object will change its gravity.

#

def set\_gravity (self, gravity):

self.\_check\_type ([box\_t, circle\_t], "change the gravity for the object")

print("pge: ", gravity)

return pgeif.set\_gravity (self.o, gravity)

#

# get\_elasticity- Pre-condition: self is a polygon or circle object.

# Post-condition: this object will be displayed in colour, c.

#

def get\_gravity (self):

self.\_check\_type ([box\_t, circle\_t, fb\_box\_t, fb\_circle\_t], "get gravity")

return pgeif.get\_gravity(self.o)

#

# get\_unit\_coord - Pre-condition: object must be a circle.

# Post-condition: return a list [x, y] representing the

# coordinate center of this object. Each x, y

# value will be between 0.0 and 1.0.

#

def get\_unit\_coord (self):

self.\_check\_type ([circle\_t], "get the unit coordinate")

return [self.get\_xpos (), self.get\_ypos ()]

#

# get\_xpos - Pre-condition: object must be a circle or polygon.

# Post-condition: return the x position of this object.

# The value will be between 0.0 and 1.0.

#

def get\_xpos (self):

self.\_check\_type ([box\_t, circle\_t], "get the xpos")

return pgeif.get\_xpos (self.o)

#

# get\_ypos - Pre-condition: object must be a circle or polygon.

# Post-condition: return the y position of this object.

# The value will be between 0.0 and 1.0.

#

def get\_ypos (self):

self.\_check\_type ([box\_t, circle\_t], "get the ypos")

return pgeif.get\_ypos (self.o)

#

# get\_xvel - Pre-condition: object must be a circle or polygon.

# Post-condition: return the x velocity of this object.

#

def get\_xvel (self):

self.\_check\_type ([box\_t, circle\_t], "get the xvel")

return pgeif.get\_xvel (self.o)

#

# get\_yvel - Pre-condition: object must be a circle or polygon.

# Post-condition: return the y velocity of this object.

#

def get\_yvel (self):

self.\_check\_type ([box\_t, circle\_t], "get the yvel")

return pgeif.get\_yvel (self.o)

#

# get\_xaccel - Pre-condition: object must be a circle or polygon.

# Post-condition: return the x accel of this object.

#

def get\_xaccel (self):

self.\_check\_type ([box\_t, circle\_t], "get the xaccel")

return pgeif.get\_xaccel (self.o)

#

# get\_yaccel - Pre-condition: object must be a circle or polygon.

# Post-condition: return the y accel of this object.

#

def get\_yaccel (self):

self.\_check\_type ([box\_t, circle\_t], "get the yaccel")

return pgeif.get\_yaccel (self.o)

#

# put\_xvel - Pre-condition: object must be a circle or polygon.

# Post-condition: assigns a float, f, as the x velocity of this

# object.

#

def put\_xvel (self, f):

# print "put\_xvel on a", self.\_name ()

self.\_check\_type ([box\_t, circle\_t], "put the xvel")

return pgeif.put\_xvel (self.o, f)

#

# put\_yvel - Pre-condition: object must be a circle or polygon.

# Post-condition: assigns a float, f, as the y velocity of this

# object.

#

def put\_yvel (self, f):

self.\_check\_type ([box\_t, circle\_t], "put the yvel")

return pgeif.put\_yvel (self.o, f)

#

# put\_xaccel - Pre-condition: object must be a circle or polygon.

# Post-condition: assigns a float, f, as the x

# acceleration of this object.

#

def put\_xaccel (self, f):

self.\_check\_type ([box\_t, circle\_t], "put the xaccel")

return pgeif.put\_xaccel (self.o, f)

#

# put\_yaccel - Pre-condition: object must be a circle or polygon.

# Post-condition: assigns a float, f, as the y

# acceleration of this object.

#

def put\_yaccel (self, f):

self.\_check\_type ([box\_t, circle\_t], "put the yaccel")

return pgeif.put\_yaccel (self.o, f)

#

# moving\_towards - Pre-condition: object must be a non fixed

# circle or polygon.

# Post-condition: returns True if this object

# is moving towards point [x, y].

# x and y must be in the range of 0.0 and 1.0.

#

def moving\_towards (self, x, y):

self.\_check\_type ([box\_t, circle\_t], "test moving\_towards")

if self.fixed:

return False

return pgeif.moving\_towards (self.o, x, y)

#

# draw - draw a spring using colour, c, and a width, w.

# It is drawn under the objects it connects.

# The spring object is returned.

#

def draw (self, c, w):

self.\_check\_not\_deleted ("spring no longer exists")

self.\_check\_type ([spring\_t], "expected a spring")

c.\_param\_colour ("the first parameter to draw is expected to be a colour")

print(self.o, c.\_get\_pgeif\_colour (), w)

pgeif.draw\_spring (self.o, c.\_get\_pgeif\_colour (), w)

return self

#

# end - draw the objects at the end of the spring with

# colour, c, when the object comes to rest.

#

def end (self, c):

self.\_check\_not\_deleted ("spring no longer exists")

self.\_check\_type ([spring\_t], "expected a spring")

c.\_param\_colour ("the parameter to end is expected to be a colour")

print(self.o, c.\_get\_pgeif\_colour ())

pgeif.end\_spring (self.o, c.\_get\_pgeif\_colour ())

print("assigned c", c.\_get\_pgeif\_colour ())

return self

#

# mid - draw a debugging frame when the spring reaches its rest point.

# The objects connected by the spring are drawn with colour, c.

#

def mid (self, c):

self.\_check\_not\_deleted ("spring no longer exists")

self.\_check\_type ([spring\_t], "expected a spring")

c.\_param\_colour ("the parameter to mid is expected to be a colour")

print(self.o, c.\_get\_pgeif\_colour ())

pgeif.mid\_spring (self.o, c.\_get\_pgeif\_colour ())

print("assigned c", c.\_get\_pgeif\_colour ())

return self

#

# when - when spring reaches a length, l, call function, p.

# p is a function with two parameters, o and e.

# o is this object.

# e is the event.

#

def when (self, l, p):

global idcount, id2func

self.\_check\_not\_deleted ("spring no longer exists")

self.\_check\_type ([spring\_t], "expected a spring")

self.springp = p

idcount += 1

id2func[idcount] = p

pgeif.when\_spring (self.o, l, idcount)

return self

#

#

#

def rotate(self, angle):

print("pge.rotate called: ", angle)

self.\_check\_type ([box\_t, circle\_t], "change the angle of rotation for the object")

return pgeif.set\_rotate (self.o, angle)

#

# Sets boolean value for visibility

#

def visibility(self, on):

print("pge.visibility called: ", on)

self.\_check\_type ([box\_t, circle\_t], "change the visibility of the object")

return pgeif.visibility (self.o, on)

#

# Gets boolean value for visibility

#

def is\_visible(self):

print("pge.is\_visible called")

self.\_check\_type ([box\_t, circle\_t], "get the visibility of the object")

return pgeif.is\_visible (self.o)

#

# \_colspace - convert a float value 0.0..1.0 into integer 0..255.

#

def \_colspace (f):

return (int)(f \* 255.0)

#

# rgb - Pre-condition: None.

# Post-condition: create a colour object from the red, green, blue triple.

# The r, g, b, values must between 0.0 and 1.0.

#

def rgb (r, g, b):

global idTOcol

# print "in rgb (", r, g, b, ")"

c = pgeif.rgb (float(r), float(g), float(b))

# print "after pgeif.rgb ->", c

o = object (colour\_t, [float(r), float(g), float(b), c])

o.\_check\_colour ()

c = pgeif.h2l (c)

idTOcol[c] = (\_colspace (r), \_colspace (g), \_colspace (b))

# print "define colour triple as:", idTOcol[c]

return o

#

# white - Pre-condition: None.

# Post-condition: return a colour object representing the colour white.

#

def white ():

c = pgeif.white ()

o = object (colour\_t, [1.0, 1.0, 1.0, c])

c = pgeif.h2l (c)

idTOcol[c] = (255, 255, 255)

return o

#

# update\_fps - default call back used by local\_fps.

# e is the event and o is the object which will be None.

#

def update\_fps (e, o):

global last\_fps, fps\_text

fn = get\_frame\_no ()

s = "fps %d" % (fn - last\_fps)

if fps\_text != None:

fps\_text.rm ()

fps\_text = text (0.8, 0.1, s, white (), 50, 1)

last\_fps = fn

local\_fps ()

#

# local\_fps - generate a simple frames per second on the screen.

#

def local\_fps ():

f = at\_time (1.0, update\_fps)

#

# \_register - register a colour pair id:ob in two dictoraries

# so we can look either up given the other.

#

def \_register (id, ob):

global id2ob, od2id

\_debugf ("registering %d\n", id)

id2ob[id] = ob

ob2id[ob] = id

#

# text - place text string, s, at position [x, y] in colour, c.

# The size will be in font size and placed in the foreground or

# background depending upon level. You are not allowed

# to place text in level zero.

#

def text (x, y, s, c, size, level):

global device, screen

c.\_param\_colour ("fourth parameter to text is expected to be a colour")

if level == 0:

\_errorf ("not allowed to place in level 0")

else:

ob = object (fb\_text\_t, [x, y, s, size, c.\_get\_pgeif\_colour ()], c, level)

\_add (ob, level)

return ob

#

# box - create a box at position, [x, y]. x and y have the range 0.0 to 1.0.

# w and h are the width and height and are in the range 0.0 to 1.0.

# c is a colour object.

# level is optional, if it is not present it defaults to zero.

# Otherwise the level determines whether it exists in the foreground

# or background.

#

def box (x, y, w, h, c, level = 0):

# print "box:", x, y, w, h, c, level

c.\_param\_colour ("fifth parameter to box is expected to be a colour")

if level == 0:

id = pgeif.box (x, y, w, h, c.\_get\_pgeif\_colour ())

# print "box colour =", c, c.\_get\_pgeif\_colour ()

ob = object (box\_t, id, c, level)

ob.set\_width (w)

\_debugf ("box ")

\_register (id, ob)

else:

ob = object (fb\_box\_t, [x, y, x+w, y, x+w, y+h, x+w, y+h, x, y+h, c.\_get\_pgeif\_colour ()], c, level)

# print "box colour =", c, c.\_get\_pgeif\_colour ()

\_add (ob, level)

return ob

#

# poly3 - create a triangle at position, [x0, y0], [x1, y1], [x2, y2] all

# values have the range 0.0 to 1.0.

# c is a colour object.

# level is optional, if it is not present it defaults to zero.

# Otherwise the level determines whether it exists in the foreground

# or background.

#

def poly3 (x0, y0, x1, y1, x2, y2, c, level = 0):

c.\_param\_colour ("seventh parameter to box is expected to be a colour")

if level == 0:

id = pgeif.poly3 (x0, y0, x1, y1, x2, y2, c.\_get\_pgeif\_colour ())

ob = object (box\_t, id, c, level)

\_debugf ("poly3 ")

\_register (id, ob)

else:

ob = object (fb\_box\_t, [x0, y0, x1, y1, x2, y2, c.\_get\_pgeif\_colour ()], c, level)

\_add (ob, level)

return ob

#

# poly4 - create a polygon of four vertices at position, [x0, y0],

# [x1, y1], [x2, y2], [x3, y3] all

# values have the range 0.0 to 1.0.

# c is a colour object.

# level is optional, if it is not present it defaults to zero.

# Otherwise the level determines whether it exists in the foreground

# or background.

# The coordinates must either be given in clockwise or anticlockwise

# order.

#

def poly4 (x0, y0, x1, y1, x2, y2, x3, y3, c, level = 0):

c.\_param\_colour ("seventh parameter to box is expected to be a colour")

if level == 0:

id = pgeif.poly4 (x0, y0, x1, y1, x2, y2, x3, y3, c.\_get\_pgeif\_colour ())

ob = object (box\_t, id, c, level)

\_debugf ("poly3 ")

\_register (id, ob)

else:

ob = object (fb\_box\_t, [x0, y0, x1, y1, x2, y2, x3, y3, c.\_get\_pgeif\_colour ()], c, level)

\_add (ob, level)

return ob

#

# \_add - adds an object at foreground/background, level.

# A level value of > 0 will be placed into the

# foreground.

# A level value of < 0 will be placed into the

# background.

#

def \_add (ob, level):

global foreground, background, levels

if level > 0:

if not (level in foreground):

foreground += [level]

foreground.sort ()

else:

if not (level in background):

background += [level]

background.sort ()

if level in levels:

levels[level] += [ob]

else:

levels[level] = [ob]

# print levels[level]

#

# \_sub - removes the object placed by the function \_add.

#

def \_sub (ob, level):

global foreground, background

if level in levels:

levels[level].remove (ob)

if level > 0:

f = []

for l in foreground:

if l in levels:

f += [l]

foreground = f

foreground.sort ()

else:

b = []

for l in background:

if l in levels:

b += [l]

background = b

background.sort ()

#

# spring - place a spring of at rest length, l, and Hook's

# value of, k, between object ob1 and object ob2.

# The objects, ob1, and, ob2, must be either

# a circle or a polygon. A spring only exists at

# level 0 (in the physics engine).

# If, l, is omitted then it is assumed that the

# spring starts at rest between, ob1, and, ob2.

# Parameter d is the damping force.

#

def spring (ob1, ob2, k, d, l = None):

ob1.\_check\_not\_deleted ("a spring attachment")

ob2.\_check\_not\_deleted ("a spring attachment")

ob1.\_check\_type ([box\_t, circle\_t], "creating a spring, first parameter")

ob2.\_check\_type ([box\_t, circle\_t], "creating a spring, second parameter")

if l == None:

l = -1

print("before pgeif.spring")

id = pgeif.spring (ob1.o, ob2.o, k, d, l)

print("after pgeif.spring =", id)

ob = object (spring\_t, id, None, 0)

\_register (id, ob)

return ob

#

# circle - place a circle at coordinate (x, y)

# The circle has a radius, r, and is filled with colour, c.

# If the level == 0 it is placed into the physics engine.

# A level < 0 is placed into the background.

# A level > 0 is placed into the foreground.

#

def circle (x, y, r, c, level = 0):

c.\_param\_colour ("fourth parameter to box is expected to be a colour")

if level == 0:

id = pgeif.circle (x, y, r, c.\_get\_pgeif\_colour ())

# print "circle id =", id

# \_debugf ("circle ")

ob = object (circle\_t, id, c, level)

\_register (id, ob)

else:

# print "circle, colour =", c

# print "pge: colour", c.\_get\_pgeif\_colour ()

ob = object (fb\_circle\_t, [x, y, r, c.\_get\_pgeif\_colour ()], c, level)

\_add (ob, level)

return ob

#

# \_unpackFract - returns three integers: w, n, d

# representing fraction.

#

def \_unpackFract (s):

b = s[0]

v = struct.unpack ("B", b)[0]

if v == 0:

return (0, 0, 0)

elif v == 1:

return (1, 0, 0)

elif v == 2:

b = s[1:17]

r = struct.unpack('!QQ', b)

return (0, r[0], r[1])

else:

b = s[1:33]

return struct.unpack('!QQQ', b)

#

# \_unpackReal

#

def \_unpackReal (s):

if len (s) >= 8:

return struct.unpack ('d', s[:8])[0]

else:

\_printf ("insufficient data passed to \_unpackReal\n")

def \_unpackCard (s):

if len (s) >= 4:

return struct.unpack ('!I', s[:4])[0]

else:

\_printf ("insufficient data passed to \_unpackCard\n")

def \_unpackCardPair (s):

if len (s) >= 8:

return [struct.unpack ('!I', s[:4])[0],

struct.unpack ('!I', s[4:8])[0]]

else:

\_printf ("insufficient data passed to \_unpackCardPair (%d bytes)\n", len (s))

def \_unpackIdPair (s):

p = \_unpackCardPair (s)

p[0] = pgeif.l2h (p[0])

p[1] = pgeif.l2h (p[1])

return p

def \_unpackPoint (s):

if len (s) >= 16:

return [\_unpackReal (s[:8]), \_unpackReal (s[8:])]

else:

\_printf ("insufficient data passed to \_unpackPoint\n")

#

# \_draw\_foreground - draws all the foreground objects in order.

#

def \_draw\_foreground ():

# print "draw foreground", foreground

if foreground != []:

for l in foreground:

# print "drawing level", l

for o in levels[l]:

o.\_draw ()

#

# \_draw\_background - draws all the foreground objects in order.

#

def \_draw\_background ():

# print "draw background", background

if background != []:

for l in background:

# print "drawing level", l

for o in levels[l]:

o.\_draw ()

#

# \_draw

#

def \_draw ():

\_draw\_background ()

\_draw\_foreground ()

\_doFlipBuffer ()

no\_event, frame\_event, collision\_event, function\_event, spring\_event, final\_event = list(range(6))

#

# the event class is used by the user through call back functions. In particular

# the user can find out the collision\_between two objects, cancel a timer event or

# query whether a timer event was cancelled.

#

class event:

def \_\_init\_\_ (self, t, d, l):

\_debugf ("creating event (data is %d bytes)\n", l)

self.\_type = t

self.\_edata = d

self.\_elength = l

self.\_fdata = None

self.\_flength = 0

self.\_cData = None

self.\_clength = 0

self.\_cancelled = False

# the following are the event data values

self.\_\_point = None

self.\_\_between = None

self.\_\_etime = 0.0

self.\_\_etype = 0

self.\_\_kind = 0

self.\_\_id = None

self.\_\_param = 0

if self.\_edata == None:

\_debugf ("final or timer event\n")

else:

\_debugf ("\*\*\*\*\*\*\*\*\*\*\* current time is %f \*\*\*\*\*\*\*\*\*\*\*\n", pgeif.get\_time ())

self.\_\_etime = \_unpackReal (self.\_edata) # 8 bytes REAL

\_debugf ("\*\*\*\*\*\*\*\*\*\*\* event time is %f \*\*\*\*\*\*\*\*\*\*\*\*\*\n", self.\_\_etime)

if t == collision\_event:

self.\_\_etype = \_unpackCard (self.\_edata[8:12]) # 4 bytes etype

self.\_\_point = \_unpackPoint (self.\_edata[12:])

self.\_\_between = \_unpackIdPair (self.\_edata[28:])

# print "assigning between values", self.\_\_between

# etype == 0 is a draw frame event

# etype == 1 two circles colliding

if self.\_\_etype == 2 or self.\_\_etype == 3:

# circle/polygon collision or polygon/polygon collision

self.\_\_kind = \_unpackCard (self.\_edata[36:])

\_debugf ("collision event created which indicates a collision in %f seconds\n", self.\_\_etime)

elif t == frame\_event:

\_debugf ("frame event %d in %f seconds\n", t, self.\_\_etime)

elif t == function\_event:

\_debugf ("function event %d in %f seconds\n", t, self.\_\_etime)

self.\_\_etype = \_unpackCard (self.\_edata[8:12]) # 4 bytes etype

self.\_\_id = \_unpackCard (self.\_edata[12:16]) # 4 bytes id

self.\_\_param = \_unpackCard (self.\_edata[16:20]) # 4 bytes unsigned int parameter

elif t == spring\_event:

\_debugf ("spring event %d in %f seconds\n", t, self.\_\_etime)

self.\_\_etype = \_unpackCard (self.\_edata[8:12]) # 4 bytes etype

self.\_\_id = \_unpackCard (self.\_edata[12:16]) # 4 bytes id

self.\_\_kind = \_unpackCard (self.\_edata[16:20]) # 4 bytes kind

else:

\_printf ("unknown event %d in %f seconds\n", t, self.\_\_etime)

def \_set\_frame\_contents (self, data, length):

self.\_fData = data

self.\_flength = length

def \_set\_colour\_contents (self, data, length):

self.\_cData = data

self.\_clength = length

def \_process (self):

global id2func

# \_printf ("current time %f, moving time forward until this event: %f\n", pgeif.get\_time (), self.\_\_etime)

pgeif.skip\_until (self.\_\_etime)

# \_printf ("current time is now %f\n", pgeif.get\_time ())

\_debugf ("\_flush\_delay\n")

\_flush\_delay ()

\_debugf ("about to call process\_event\n")

pgeif.process\_event ()

\_debugf ("find out which event\n")

if self.\_type == frame\_event:

self.\_handle\_frame\_buffer ()

elif self.\_type == collision\_event:

\_debugf ("collision event seen, in %f seconds\n", self.\_\_etime)

# pgeif.skip\_until (self.\_\_etime)

self.\_handle\_frame\_buffer ()

\_collision (self.\_between (), self)

elif self.\_type == function\_event:

# print "\_process found timer\_event", self.\_\_id

i = self.\_\_id

p = self.\_\_param

if i in id2func:

# print "function", i, "about to be called with parameter", p

if p == 0:

id2func[i] (self, None)

else:

id2func[i] (self, id2ob[pgeif.l2h (p)])

# print "function", i, "finished"

else:

# print "function", i, "has been cancelled"

pass

def \_handle\_frame\_buffer (self):

cData = pgeif.get\_cbuf ()

\_debugf ("cData len = %d\n", len (cData))

self.\_set\_colour\_contents (cData, len (cData))

fData = pgeif.get\_fbuf ()

\_debugf ("fData len = %d\n", len (fData))

self.\_set\_frame\_contents (fData, len (fData))

\_draw\_frame (self.\_cData, self.\_clength,

self.\_fData, self.\_flength)

pgeif.empty\_fbuffer ()

pgeif.empty\_cbuffer ()

def \_check (self, et):

if self.\_type != et:

\_printf ("fatal error, unexpected event type\n")

sys.exit (1)

def \_between (self):

global id2ob

self.\_check (collision\_event)

# returns the two object ids of the colliding objects

\_debugf ("id0 = %d, id1 = %d\n", self.\_\_between[0], self.\_\_between[1])

ob1 = id2ob[self.\_\_between[0]]

ob2 = id2ob[self.\_\_between[1]]

return [ob1, ob2]

def \_get\_time (self):

return self.\_\_etime

#

# collision\_between - Pre-condition: event object must a collision event.

# Post-condition: returns a list of two objects which

# are in collision.

#

def collision\_between (self):

return self.\_between ()

#

# cancel - Pre-condition: event must be a timer event.

# Post-condition: timer event is cancelled.

#

def cancel (self):

self.\_cancelled = True

#

# was\_cancelled - Pre-condition: event must be a timer event.

# Post-condition: returns True if this event was cancelled.

#

def was\_cancelled (self):

return self.\_cancelled

#

# get\_timer\_id - Pre-condition: event must be a timer event

# Post-condition: timer id is returned.

#

def get\_timer\_id (self):

return self.\_\_id

#

# \_get\_next\_event - returns a next event object which has

# contains any relevant information from

# the physics engine.

#

def \_get\_next\_event ():

global device

\_debugf ("\_get\_next\_event\n")

\_setDefaultDevice ()

if pgeif.is\_collision ():

\_debugf ("pgeif.is\_collision\n")

\_debugf ("pgeif.get\_ebuf\n")

eData = pgeif.get\_ebuf ()

\_debugf ("event (...\n")

return event (collision\_event, eData, len (eData))

elif pgeif.is\_frame ():

\_debugf ("pgeif.is\_frame\n")

\_debugf ("pgeif.get\_ebuf\n")

eData = pgeif.get\_ebuf ()

# print "testing -> ", \_unpackReal (eData)

return event (frame\_event, eData, len (eData))

elif pgeif.is\_function ():

# \_printf ("pgeif.is\_function\n")

eData = pgeif.get\_ebuf ()

return event (function\_event, eData, len (eData))

elif pgeif.is\_spring ():

\_debugf ("pgeif.is\_spring\n")

\_debugf ("pgeif.get\_ebuf\n")

eData = pgeif.get\_ebuf ()

\_debugf ("event (...\n")

return event (spring\_event, eData, len (eData))

else:

\_printf ("fatal error: unknown event type (terminating simulation)\n")

sys.exit (1)

return event (no\_event, None, 0)

#

# \_collision - call the \_collision method for every object

# in the, between, list.

#

def \_collision (between, event):

for o in between:

o.\_collision (between, event)

#

# \_process - call the \_process method for in the object, pe.

#

def \_process (pe):

pe.\_process ()

#

# \_add\_relative - adds, r a list [time, event]

# to the pge\_event\_queue. It maintains

# their position using a relative ordered queue.

#

def \_add\_relative (r):

global pge\_event\_queue

if pge\_event\_queue == []:

# catch the easy case early

pge\_event\_queue = [r]

return

abs\_time = r[0]

acc\_time = pge\_event\_queue[0][0]

if abs\_time < acc\_time:

# at the front of the queue

pge\_event\_queue[0][0] -= abs\_time

pge\_event\_queue = [r] + pge\_event\_queue

return

i = 0

while True:

if i == len (pge\_event\_queue)-1:

# end of queue

abs\_time -= acc\_time

pge\_event\_queue += [[abs\_time, r[1]]]

return

i += 1

if acc\_time + pge\_event\_queue[i][0] < abs\_time:

acc\_time += pge\_event\_queue[i][0]

else:

# r needs to be before i

r = [abs\_time - acc\_time, r[1]]

pge\_event\_queue = pge\_event\_queue[:i] + [r] + pge\_event\_queue[i:]

# and alter relative value of, i

acc\_time += pge\_event\_queue[i][0]

if i < len (pge\_event\_queue)-1:

pge\_event\_queue = pge\_event\_queue[:i] + [[acc\_time-abs\_time, pge\_event\_queue[i][1]]] + pge\_event\_queue[i+1:]

else:

pge\_event\_queue = pge\_event\_queue[:i] + [[acc\_time-abs\_time, pge\_event\_queue[i][1]]]

return

pge\_event\_queue = []

#

# \_display\_element - debugging function to display an event element.

#

def \_display\_element (e, t):

print("[", e[0], "ms ", end=' ')

if e[1].\_type == frame\_event:

print("displayframe", end=' ')

elif e[1].\_type == collision\_event:

print("collision", end=' ')

elif e[1].\_type == function\_event:

print("timer", end=' ')

else:

print("final", end=' ')

print(" at", e[0] + (int) (t \* 1000.0), "ms", end=' ')

print("], ", end=' ')

#

# \_display\_event\_queue - debugging function to display the entire event queue.

#

def \_display\_event\_queue (q):

if q == []:

print("event queue is empty")

else:

print("event queue: ")

t = pgeif.get\_time ()

for e in q:

\_display\_element (e, t)

t += e[1].\_get\_time ()

print("")

prev\_event\_time = 0.0

#

# \_post\_event - places an event, e, at time, t, onto the event queue.

#

def \_post\_event (e, t):

global pge\_event\_queue, debugging, prev\_event\_time

if t != -1:

t = (int) (t \* 1000.0)

\_add\_relative ([t, e])

if debugging:

\_display\_event\_queue (pge\_event\_queue)

c = pge\_event\_queue[0][0] + (int) (pgeif.get\_time () \* 1000.0)

# if c < prev\_event\_time:

# \_printf ("clock skew detected\n")

prev\_event\_time = c

return e

#

# \_wait\_for\_event - waits until an event occurs. If a queued event

# exists remove it. It also checks the pygame event

# queue.

#

def \_wait\_for\_event ():

global pge\_event\_queue, slow\_down\_factor, device, \_record, debugging

if debugging:

print("\_wait\_for\_event, pge\_event\_queue =")

\_display\_event\_queue (pge\_event\_queue)

if device == pyg\_d:

pygame.event.set\_allowed (None)

pygame.event.set\_allowed (allowed\_events)

if pge\_event\_queue == []:

return [pygame.event.wait()] + pygame.event.get()

ms = pge\_event\_queue[0][0]

if \_record:

\_record\_time (ms)

if ms == 0:

# adding [None] to the list indicates an immediate pge event

# we obtain all pygame events to avoid starvation of input

return pygame.event.get() + [None]

# \_printf ("setting timer in pygame to %d ms\n", ms)

pygame.time.set\_timer (USEREVENT+1, (int) (ms \* slow\_down\_factor))

return [pygame.event.wait()] + pygame.event.get()

#

# \_finish\_event - creates and returns a final\_event.

#

def \_finish\_event ():

return event (final\_event, None, 0)

#

# at\_time - informs the physics engine to create a call to function, p, at

# time, t, in the future. It returns a integer reference for the

# timed function.

# Pre-condition: t is a time seconds (float) in the future.

# p, is a function which takes two parameters

# the first parameter is the event and the

# second is an unused parameter.

# The second parameter is only there to allow

# coexistance with other call back functions.

# Post-condition: function p is placed into the timer list

# and a timer id (integer) is returned.

#

def at\_time (t, p):

global idcount, id2func, slow\_down\_factor

idcount += 1

pgeif.create\_function\_event (t / slow\_down\_factor, idcount, 0)

id2func[idcount] = p

return idcount

#

# at\_cancel - Pre-condition: the integer reference for the timed function.

# Post-condition: cancel the timed function.

#

def at\_cancel (i):

global id2func

if i in id2func:

del id2func[i]

else:

error ("at\_cancel cannot delete function %d as it no longer exists\n", i)

#

# record - record the game. A file output.raw will be created which

# can be post processed by pgeplayback.

# Pre-condition: None.

# Post-condition: the record flag is set and the game will

# be recorded.

#

def record ():

global \_record

\_record = True

#

# \_draw\_frame - draws a frame on the chosen device.

#

def \_draw\_frame (cdata, clength, fdata, flength):

global device, frame\_no

frame\_no += 1

if device == pyg\_d:

\_pyg\_draw\_frame (cdata, clength, fdata, flength)

else:

\_batch\_draw\_frame (cdata, clength, fdata, flength)

#

# \_pyg\_draw\_frame - draws a frame on the pygame display.

#

def \_pyg\_draw\_frame (cdata, clength, fdata, flength):

global nextFrame, call, \_record

# \_printf ("enter \_pyg\_draw\_frame: in frame %d (length = %d bytes)\n", nextFrame, flength+clength)

# \_printf ("drawing background\n")

if \_record:

\_begin\_record\_frame (cdata, clength, fdata, flength)

elif flength > 0:

\_draw\_background ()

f = \_myfile (cdata + fdata)

while f.left () >= 3:

header = struct.unpack ("3s", f.read (3))[0]

header = header[:2]

if header in call:

f = call[header] (f)

else:

print("not understood header =", header)

sys.exit (1)

# \_printf ("drawing foreground\n")

if flength > 0:

\_draw\_foreground ()

if \_record:

\_end\_record\_frame ()

if flength > 0:

\_doFlipBuffer () # flipping the buffer for an empty frame looks ugly

# print "end of draw"

nextFrame += 1

\_debugf ("moving onto frame %d\n", nextFrame)

def \_check\_opened ():

global opened, output

if not opened:

opened = True

output = open ("output.raw", "wb")

def \_begin\_record\_frame (cdata, clength, fdata, flength):

global opened, output, nextFrame

\_check\_opened ()

# output.write (struct.pack ("3s", "fn"))

output.write (struct.pack("3s", "fn".encode('ascii'))) # frame note

\_emit\_card (nextFrame)

if clength > 0:

\_debugf ("writing colour data length = %d bytes\n", clength)

output.write (cdata)

if flength > 0:

\_draw\_background ()

\_debugf ("writing frame data length = %d bytes\n", flength)

output.write (fdata)

# c.sendall(b'Thank you for connecting')

# output = 'Thank you for connecting'

# c.sendall(output.encode('utf-8'))

else:

pass

# \_printf ("length of zero!!\n")

# sys.exit (2)

def \_end\_record\_frame ():

output.write (struct.pack ("3s", "fb".encode ("utf-8"))) # flip buffer

#

# \_record\_time - records the time delay.

#

def \_record\_time (ms):

global output, slow\_down\_factor

ms = ((float) (ms)) \* slow\_down\_factor / 1000.0

# print "recording time", ms

output.write (struct.pack ("3s", "sl".encode ("utf-8"))) # sleep

\_emit\_double (ms)

#

# \_batch\_draw\_frame - records the frame.

#

def \_batch\_draw\_frame (cdata, clength, fdata, flength):

global opened, output, nextFrame

\_debugf ("\_batch\_draw\_frame\n")

if fdata is None:

\_printf ("no data in the frame!\n")

sys.exit (1)

if not opened:

opened = True

output = open ("output.raw", "wb")

nextFrame = 1

\_begin\_record\_frame (cdata, clength, fdata, flength)

\_end\_record\_frame ()

nextFrame += 1

#

# gravity - Pre-condition: None.

# Post-condition: turn on pge world gravity, if parameter

# is missing the default of 9.81ms^2 is used.

#

def gravity (value=-9.81):

pgeif.gravity (value)

#

# get\_font - Pre-condition: None.

# Post-condition: return the font corresponding

# to size pixels height.

#

def get\_font (size):

global font, font\_size

if size == font\_size:

return font

else:

font\_size = size

font = pygame.font.Font (None, font\_size)

return font

#

# \_init\_screen - initialise the pygame screen.

#

def \_init\_screen ():

global resolution, fullscreen, screen\_initialised, backcanvas, screen, program\_name, version\_number

if not screen\_initialised:

pygame.init ()

if fullscreen:

screen = pygame.display.set\_mode (resolution, FULLSCREEN)

else:

screen = pygame.display.set\_mode (resolution)

screen\_initialised = True

pygame.display.set\_caption (program\_name + ' ' + version\_number)

backcanvas = pygame.Surface (screen.get\_size ())

backcanvas = backcanvas.convert ()

backcanvas.fill (Black)

#

# register\_handler - Pre-condition: None.

# Post-condition: call, function, if any event in pyeventlist

# occurs.

#

def register\_handler (function, pyeventlist):

global pyevent2func, allowed\_events

for e in pyeventlist:

pyevent2func[e] = function

allowed\_events += pyeventlist

#

# deregister\_handler - Pre-condition: None.

# Post-condition: remove the handlers for pyeventlist.

#

def deregister\_handler (pyeventlist):

global pyevent2func, allowed\_events

for e in pyeventlist:

del pyevent2func[e]

allowed\_events.remove (e)

#

# runpy - Pre-condition: all objects have been initialised correctly in the 2D world.

# Post-condition: runs pge for time, t, seconds and also

# process the pygame events the objects appear in the Pygame screen.

#

def runpy (t=-1):

global pge\_event\_queue

\_init\_screen ()

pgeif.use\_time\_delay (False)

cData = pgeif.get\_cbuf ()

fData = pgeif.get\_fbuf ()

\_check\_opened ()

\_draw\_frame (cData, len (cData), fData, len (fData))

pgeif.empty\_fbuffer ()

pgeif.empty\_cbuffer ()

if pge\_event\_queue == []:

# no events yet, so collect the next from the physics engine

ev = \_get\_next\_event ()

nev = \_post\_event (ev, ev.\_get\_time ())

# always add the final event which is the only way to finish the while loop

fin = \_post\_event (\_finish\_event (), t)

while True:

for e in \_wait\_for\_event ():

if (e == None) or (e.type == USEREVENT+1):

# immediate pge event pending or user event has fired

# take the event off queue

pe = pge\_event\_queue [0][1]

pge\_event\_queue = pge\_event\_queue [1:]

if pe == fin:

# the finish event, we're done and out of here

return

\_process (pe)

ev = \_get\_next\_event ()

nev = \_post\_event (ev, ev.\_get\_time ())

elif e.type in pyevent2func:

pyevent2func[e.type] (e)

#

# runbatch - Pre-condition: all objects have been initialised correctly in the 2D world.

# Post-condition: runs pge for time, t. If t < 0.0 then simulate for 30.0 seconds max.

# Nothing is rendered visually, but the graphics can be directed to a file using the

# record function.

#

def runbatch (t):

if t < 0.0:

t = 30.0

\_debugf ("runbatch (%f)\n", t)

if pgeif.check\_objects ():

cData = pgeif.get\_cbuf ()

fData = pgeif.get\_fbuf ()

\_draw\_frame (cData, len (cData), fData, len (fData))

pgeif.empty\_fbuffer ()

pgeif.empty\_cbuffer ()

nev = \_get\_next\_event ()

acc = 0.0

while acc+nev.\_get\_time () < t:

old = acc

acc = acc + nev.\_get\_time ()

delay (nev.\_get\_time ())

if int(acc) != int(old):

\_printf ("%d/%d seconds completed %d%%\n", int (acc), int (t), int (acc\*100.0/t))

\_process (nev)

nev = \_get\_next\_event ()

else:

\_errorf ("'runbatch' detected a non fixed object has no mass, all moving objects must have a mass\n")

#

# display\_set\_mode - Pre-condition: this must be called before pge.run.

# The parameter is a list [x, y].

# Current limitation the y parameter

# is overwritten by the x value.

# Post-condition: the resolution will be used when pge.run.

# is called.

#

def display\_set\_mode (r):

global resolution

r[1] = r[0]

resolution = r

#

# display\_fullscreen - Pre-condition: this must be called before pge.run.

# Post-condition: configure the Pygame to use fullscreen.

#

def display\_fullscreen (b):

global fullscreen

fullscreen = b

#

# fps - Pre-condition: this must be called before pge.run.

# Post-condition: the game engine will render the world at, f, frames per second.

#

def fps (f):

global framesPerSecond

framesPerSecond = f

#

# get\_frame\_no - return the current frame number.

#

def get\_frame\_no ():

global frame\_no

return frame\_no

#

# run - Pre-condition: all objects and any screen resolution must be configured.

# Post-condition: runs pge for time, t, seconds and also

# process the pygame events

#

def run (t=-1):

global device

\_setDefaultDevice ()

pgeif.fps (framesPerSecond)

if device == pyg\_d:

runpy (t)

else:

runbatch (t)

def \_setDevice (d):

global device

if device == None:

device = d

if device == pyg\_d:

pygame.init ()

pgeif.use\_buffer ()

else:

\_printf ("cannot change device once pge has started\n")

def \_setDefaultDevice ():

global device

if device == None:

device = pyg\_d

pgeif.use\_buffer ()

#

# batch - Pre-condition: None.

# Post-condition: configures pge to run in batch mode, without any display

# and without any pygame. No input events are allowed in this mode.

#

def batch ():

\_setDevice (batch\_d)

#

# interactive - Pre-condition: None.

# Post-condition: configures pge to use pygame and it allows interactivity from

# the python pygame event queue.

#

def interactive ():

\_setDevice (pyg\_d)

#

# finish\_record - Pre-condition: record must have been called.

# Post-condition: finish recording the game and flush the record file.

#

def finish\_record ():

global output, opened

if opened:

output.close ()

opened = False

def \_load\_sound (name):

class \_NoneSound:

def play(self):

pass

if not pygame.mixer or not pygame.mixer.get\_init():

return \_NoneSound()

try:

sound = pygame.mixer.Sound(name)

except pygame.error as message:

print('cannot load sound file:', name)

return \_NoneSound()

return sound

def play (name):

global output

\_flush\_delay ()

if device == pyg\_d:

s = \_load\_sound (name)

s.play ()

else:

output.write (struct.pack ("3s", "ps".encode ("utf-8")))

output.write (name)

output.write ('\0')

#

# message - write out text to the output.

#

def message (text):

output.write (struct.pack ("3s", "ms".encode ("utf-8")))

output.write (text)

#

# turn the drawing of collision frames on or off.

#

# actual: determines whether an extra frame is generated

# at the time of actual collision.

# predict: draws a frame predicting the next collision.

# It will show the points predicted to collide.

#

def draw\_collision (actual, predict):

pgeif.draw\_collision (actual, predict)

#

# collision\_colour - if draw\_collision is called and

# with its first parameter, actual = True, then

# the objects in collision will have colour, c.

#

def collision\_colour (c):

c.\_param\_colour ("first parameter to collision\_colour is expected to be a colour")

pgeif.set\_collision\_colour (c.\_get\_pgeif\_colour ())

#

# dump\_world - this is a debugging routine which allows users to

# obtain a textual dump of the objects in the

# physics engine.

#

def dump\_world ():

pgeif.dump\_world ()

#

# \_flush\_delay - write out or implement the collected delay time.

#

def \_flush\_delay ():

global lastDelay

if lastDelay > 0.0:

\_debugf ("delay of %f\n", lastDelay)

if device == pyg\_d:

time.sleep (lastDelay)

else:

output.write (struct.pack ("3s", "sl".encode ("utf-8")))

output.write (struct.pack ("d", lastDelay))

lastDelay = 0.0

#

# delay - introduce a delay for, t.

#

def delay (t):

global lastDelay

lastDelay += t

#

# slow\_down - slow down the physics engine by a factor of, t.

#

def slow\_down (t):

global slow\_down\_factor

slow\_down\_factor = t

def \_readShort (f):

b = f.read (2)

c = struct.unpack ('!H', b)[0]

return f, c

def \_toCol (f):

return toFloat (f)\*255

#

# \_readFract - returns three integers: w, n, d

# representing fraction.

#

def \_readFract (f):

b = f.read (1)

v = struct.unpack ("B", b)[0]

if v == 0:

return f, (0, 0, 0)

elif v == 1:

return f, (1, 0, 0)

elif v == 2:

b = f.read (8\*2)

r = struct.unpack('!QQ', b)

return f, (0, r[0], r[1])

else:

b = f.read (8\*3)

return f, struct.unpack('!QQQ', b)

#

# \_mults -

#

def \_mults (s, f):

if s == 0:

return 0

if f[1] == 0 or f[2] == 0:

return int (f[0]\*s)

return int (f[0]+f[1]\*s/f[2])

def toFloat (f):

if f[1] == 0 or f[2] == 0:

return float(f[0])

return float(f[0]) + float(f[1])/float(f[2])

def \_doRegisterColour (f):

global idTOcol, debugging

f, c = \_readShort (f)

f, rf = \_readFract (f)

f, gf = \_readFract (f)

f, bf = \_readFract (f)

if debugging:

print(rf, gf, bf)

r = \_toCol (rf)

g = \_toCol (gf)

b = \_toCol (bf)

idTOcol[c] = (r, g, b)

return f

def \_doExit (f):

\_debugf ("doExit called\n")

sys.exit (0)

return f

#

# \_readColourRaw - returns the file and colour id (short).

#

def \_readColourRaw (f):

f, c = \_readShort (f)

return f, c

#

# \_readColour - returns the file and colour triple.

#

def \_readColour (f):

f, c = \_readColourRaw (f)

\_debugf ("colour value %d\n", c)

col = idTOcol[c]

return f, col

#

# drawFillPolygon -

#

def \_doDrawFillPolygon (f):

global screen, debugging

\_debugf ("doDrawFillPolygon\n")

f, n = \_readShort (f)

l = []

for i in range (n):

f, xf = \_readFract (f)

f, yf = \_readFract (f)

if debugging:

print(xf, yf, end=' ')

x = \_mults (resolution[0], xf)

y = \_mults (resolution[1], yf)

l += [[x, flip (y)]]

f, c = \_readColour (f)

if debugging:

print("drawFillPolygon (colour =", c, " l =", l, ")")

pygame.draw.polygon (screen, c, l, 0)

return f

#

# flip - returns the y value flipped against the resolution.

#

def flip (y):

global resolution

return min (resolution[0], resolution[1])-y

#

# doDrawFillCircle -

#

def \_doDrawFillCircle (f):

global screen, debugging

f, xf = \_readFract (f)

f, yf = \_readFract (f)

f, rf = \_readFract (f)

x = \_mults (resolution[0], xf)

y = \_mults (resolution[1], yf)

r = \_mults (resolution[0], rf)

f, c = \_readColour (f)

\_debugf("circle x = %d y = %d, r = %d\n", x, y, r)

if debugging:

print(" colour =", c)

pygame.draw.circle (screen, c, (x, flip (y)), r, 0)

return f

def \_wait\_for\_n ():

\_printf ("press 'n' to continue\n")

while True:

e = pygame.event.wait ()

# print e

if e.type == KEYDOWN and e.key == K\_n:

\_printf (" ... continuing\n")

return

#

# flipBuffer - flips the screen buffer.

#

def \_doFlipBuffer ():

global background, screen, nextFrame, backcanvas, program\_name

# \_printf ("doFlipBuffer called for frame (%d)\n", nextFrame)

pygame.display.set\_caption (program\_name + ' ' + version\_number + ' (%d)' % (nextFrame))

pygame.display.flip ()

screen.blit (backcanvas, (0, 0))

# \_wait\_for\_n ()

def \_doSleep (f):

global lastDelay

f, t = \_readReal (f)

# lastDelay += t

return f

def \_readReal (f):

b = f.read (8)

return f, struct.unpack ("d", b)[0]

#

# doDrawPolygon -

#

def \_doDrawPolygon (f):

global debugging, screen

f, n = \_readShort (f)

l = []

\_debugf ("drawPolygon: %d", n)

for i in range (n):

f, xf = \_readFract (f)

f, yf = \_readFract (f)

if debugging:

print(xf, yf, end=' ')

x = \_mults (resolution[0], xf)

y = \_mults (resolution[1], yf)

l += [[x, flip(y)]]

f, t = \_readFract (f)

if debugging:

print("draw polygon", l, "thickness", t)

# pygame.draw.polygon (screen, c, l, 0)

return f

def \_doMessage (f):

text = ""

b = f.read (1)

while int(b) != 0:

text += b

b = f.read (1)

\_printf ("Frame [%d]: %s\n", nextFrame, text)

#

# \_doPass - a nop.

#

def \_doPass (f):

return f

call[b'rc'] = \_doRegisterColour

call[b'dp'] = \_doDrawPolygon

call[b'dP'] = \_doDrawFillPolygon

# call['dc'] = doDrawCircle

call[b'dC'] = \_doDrawFillCircle

call[b'fb'] = \_doPass

# call['fr'] = doFramesPerSecond

call[b'ex'] = \_doExit

call[b'sl'] = \_doSleep

# call['ps'] = doPlay

# call['fn'] = doFrameNote

call[b'ms'] = \_doMessage

#

# coordinate geometry utilities

#

#

# pyg\_to\_unit\_coord - inputs : v a list of two integers in the range 0..xresolution, 0..yresolution.

# returns: a list of two floating point numbers between 0.0 and 1.0

def pyg\_to\_unit\_coord (v):

global resolution

if resolution == None:

\_errorf ("you must assign the screen resolution with a call to 'display\_set\_mode' before calling 'pyg\_to\_unit\_coord'\n")

if len (v) == 2:

return [(float) (v[0]) / (float) (resolution[0]),

(float) (resolution[1] - v[1]) / (float) (resolution[1])]

else:

\_errorf ("'pyg\_to\_unit\_coord' expects a list of two integers\n")

#

# normalise - input a vector [x, y]

# return the vector after it has been normalised.

#

def normalise (v):

x = (float) (v[0])

y = (float) (v[1])

l = magnitude ([x, y])

return [x/l, y/l]

#

# magnitude - return the modulus or magnitude of a vector or

# the Pythagorean value of the vector.

#

def magnitude (v):

return math.sqrt (v[0]\*v[0] + v[1]\*v[1])

#

# sub\_coord - returns the vector a - b.

#

def sub\_coord (a, b):

return [a[0]-b[0], a[1]-b[1]]

**i/pgeif.i**

/\* this needs to be maintained \*/

%module pgeif

%include exception.i

%exception {

try {

$action

} catch (int i) {

return NULL;

}

}

%include cstring.i

%cstring\_output\_allocate\_size(char \*\*s, int \*slen, );

%{

#define SWIG\_PYTHON\_STRICT\_BYTE\_CHAR

extern "C" void get\_cbuf (char \*\*s, int \*slen);

extern "C" void get\_ebuf (char \*\*s, int \*slen);

extern "C" void get\_fbuf (char \*\*s, int \*slen);

extern "C" void empty\_cbuffer (void);

extern "C" void empty\_fbuffer (void);

extern "C" void batch (void);

extern "C" void use\_buffer (void);

extern "C" void use\_time\_delay (unsigned int on) ;

extern "C" void draw\_collision (unsigned int actual, unsigned int predict);

extern "C" void set\_collision\_colour (unsigned int c);

extern "C" void dump\_world (void);

extern "C" unsigned int check\_objects (void);

extern "C" unsigned int rm (unsigned int id);

extern "C" double get\_time (void);

extern "C" double time\_until (void);

extern "C" unsigned int is\_frame (void);

extern "C" unsigned int is\_collision (void);

extern "C" unsigned int is\_function (void);

extern "C" unsigned int is\_spring (void);

extern "C" void create\_function\_event (double t, unsigned int id, unsigned int param);

extern "C" void process\_event (void);

extern "C" unsigned int rotate (unsigned int id, double angle);

extern "C" unsigned int is\_visible (unsigned int id);

extern "C" unsigned int visibility (unsigned int id, unsigned int on);

extern "C" unsigned int accel (unsigned int id, double ax, double ay);

extern "C" unsigned int velocity (unsigned int id, double vx, double vy);

extern "C" unsigned int spring (unsigned int id1, unsigned int id2, double k, double d, double l);

extern "C" unsigned int circle (double x0, double y0, double radius, unsigned int c);

extern "C" unsigned int fix (unsigned int id);

extern "C" unsigned int unfix (unsigned int id);

extern "C" unsigned int is\_fixed (unsigned int id);

extern "C" unsigned int mass (unsigned int id, double m);

extern "C" double get\_mass (unsigned int id);

extern "C" double get\_gravity (unsigned int id);

extern "C" double set\_gravity (unsigned int id, double g);

extern "C" unsigned int poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, unsigned int c);

extern "C" unsigned int poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, unsigned int c);

extern "C" unsigned int poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, unsigned int c);

extern "C" unsigned int poly3 (double x0, double y0, double x1, double y1, double x2, double y2, unsigned int c);

extern "C" unsigned int box (double x0, double y0, double i, double j, unsigned int c);

extern "C" double get\_xpos (unsigned int id);

extern "C" double get\_ypos (unsigned int id);

extern "C" double get\_xvel (unsigned int id);

extern "C" double get\_yvel (unsigned int id);

extern "C" double get\_xaccel (unsigned int id);

extern "C" double get\_yaccel (unsigned int id);

extern "C" double set\_rotate (unsigned int id, double angle);

extern "C" void apply\_impulse (unsigned int id, double x, double y, double m);

extern "C" int moving\_towards (unsigned int id, double x, double y);

extern "C" void put\_xvel (unsigned int id, double d);

extern "C" void put\_yvel (unsigned int id, double d);

extern "C" void put\_xaccel (unsigned int id, double d);

extern "C" void put\_yaccel (unsigned int id, double d);

extern "C" void set\_colour (unsigned int id, unsigned int c);

extern "C" void set\_elasticity (unsigned int id, double elasticity);

extern "C" double get\_elasticity (unsigned int id);

extern "C" void draw\_spring (unsigned int id, unsigned int c, double w);

extern "C" void end\_spring (unsigned int id, unsigned int c);

extern "C" void mid\_spring (unsigned int id, unsigned int c);

extern "C" void when\_spring (unsigned int id, double length, unsigned int func);

extern "C" void gravity (double g);

extern "C" unsigned int purple (void);

extern "C" unsigned int blue (void);

extern "C" unsigned int green (void);

extern "C" unsigned int red (void);

extern "C" unsigned int black (void);

extern "C" unsigned int white (void);

extern "C" unsigned int rgb (double r, double g, double b);

extern "C" unsigned int l2h (unsigned int id);

extern "C" unsigned int h2l (unsigned int id);

extern "C" double skip\_until (double t);

extern "C" void fps (double t);

%}

extern "C" void empty\_cbuffer (void);

extern "C" void empty\_fbuffer (void);

extern "C" void batch (void);

extern "C" void use\_buffer (void);

extern "C" void use\_time\_delay (unsigned int on) ;

extern "C" void draw\_collision (unsigned int actual, unsigned int predict);

extern "C" void set\_collision\_colour (unsigned int c);

extern "C" void dump\_world (void);

extern "C" unsigned int check\_objects (void);

extern "C" unsigned int rm (unsigned int id);

extern "C" double get\_time (void);

extern "C" double time\_until (void);

extern "C" unsigned int is\_frame (void);

extern "C" unsigned int is\_collision (void);

extern "C" unsigned int is\_function (void);

extern "C" unsigned int is\_spring (void);

extern "C" void create\_function\_event (double t, unsigned int id, unsigned int param);

extern "C" void process\_event (void);

extern "C" unsigned int rotate (unsigned int id, double angle);

extern "C" unsigned int is\_visible (unsigned int id);

extern "C" unsigned int visibility (unsigned int id, unsigned int on);

extern "C" unsigned int accel (unsigned int id, double ax, double ay);

extern "C" unsigned int velocity (unsigned int id, double vx, double vy);

extern "C" unsigned int spring (unsigned int id1, unsigned int id2, double k, double d, double l);

extern "C" unsigned int circle (double x0, double y0, double radius, unsigned int c);

extern "C" unsigned int fix (unsigned int id);

extern "C" unsigned int unfix (unsigned int id);

extern "C" unsigned int is\_fixed (unsigned int id);

extern "C" unsigned int mass (unsigned int id, double m);

extern "C" double get\_mass (unsigned int id);

extern "C" double get\_gravity (unsigned int id);

extern "C" double set\_gravity (unsigned int id, double g);

extern "C" unsigned int poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, unsigned int c);

extern "C" unsigned int poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, unsigned int c);

extern "C" unsigned int poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, unsigned int c);

extern "C" unsigned int poly3 (double x0, double y0, double x1, double y1, double x2, double y2, unsigned int c);

extern "C" unsigned int box (double x0, double y0, double i, double j, unsigned int c);

extern "C" double get\_xpos (unsigned int id);

extern "C" double get\_ypos (unsigned int id);

extern "C" double get\_xvel (unsigned int id);

extern "C" double get\_yvel (unsigned int id);

extern "C" double get\_xaccel (unsigned int id);

extern "C" double get\_yaccel (unsigned int id);

extern "C" double set\_rotate (unsigned int id, double angle);

extern "C" void apply\_impulse (unsigned int id, double x, double y, double m);

extern "C" int moving\_towards (unsigned int id, double x, double y);

extern "C" void put\_xvel (unsigned int id, double d);

extern "C" void put\_yvel (unsigned int id, double d);

extern "C" void put\_xaccel (unsigned int id, double d);

extern "C" void put\_yaccel (unsigned int id, double d);

extern "C" void set\_colour (unsigned int id, unsigned int c);

extern "C" void set\_elasticity (unsigned int id, double elasticity);

extern "C" double get\_elasticity (unsigned int id);

extern "C" void draw\_spring (unsigned int id, unsigned int c, double w);

extern "C" void end\_spring (unsigned int id, unsigned int c);

extern "C" void mid\_spring (unsigned int id, unsigned int c);

extern "C" void when\_spring (unsigned int id, double length, unsigned int func);

extern "C" void gravity (double g);

extern "C" unsigned int purple (void);

extern "C" unsigned int blue (void);

extern "C" unsigned int green (void);

extern "C" unsigned int red (void);

extern "C" unsigned int black (void);

extern "C" unsigned int white (void);

extern "C" unsigned int rgb (double r, double g, double b);

extern "C" unsigned int l2h (unsigned int id);

extern "C" unsigned int h2l (unsigned int id);

extern "C" double skip\_until (double t);

extern "C" void get\_cbuf (char \*\*s, int \*slen);

extern "C" void get\_ebuf (char \*\*s, int \*slen);

extern "C" void get\_fbuf (char \*\*s, int \*slen);

extern "C" void fps (double t);

**c/pgeif.c**

/\* do not edit automatically generated by mc from pgeif. \*/

/\* This file is part of GNU Modula-2.

GNU Modula-2 is free software; you can redistribute it and/or modify it under

the terms of the GNU General Public License as published by the Free

Software Foundation; either version 3, or (at your option) any later

version.

GNU Modula-2 is distributed in the hope that it will be useful, but WITHOUT ANY

WARRANTY; without even the implied warranty of MERCHANTABILITY or

FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License

for more details.

You should have received a copy of the GNU General Public License along

with gm2; see the file COPYING. If not, write to the Free Software

Foundation, 51 Franklin Street, Fifth Floor,

Boston, MA 02110-1301, USA. \*/

# if !defined (PROC\_D)

# define PROC\_D

typedef void (\*PROC\_t) (void);

typedef struct { PROC\_t proc; } PROC;

# endif

# if !defined (FALSE)

# define FALSE (1==0)

# endif

#include <string.h>

#include <limits.h>

# include "GStorage.h"

# include "Gmcrts.h"

# include "sys/cdefs.h"

#define \_pgeif\_H

#define \_pgeif\_C

# include "GStorage.h"

# include "GdeviceIf.h"

# include "GtwoDsim.h"

# include "GSYSTEM.h"

# include "GIndexing.h"

# include "GFractions.h"

# include "Glibc.h"

# include "Groots.h"

# define debugging FALSE

# define tracing FALSE

typedef struct \_T1\_r \_T1;

typedef \_T1 \*def;

typedef enum {IncorrectType, IdOutOfBounds, ValueOutOfRange} ExceptionKind;

typedef enum {colour, object} TypeOfDef;

struct \_T1\_r {

TypeOfDef type;

unsigned int definition;

};

static Indexing\_Index listOfDefs;

/\*

rgb - make a colour object using red, blue and green components.

The colour object is returned.

\*/

unsigned int rgb (double r, double g, double b);

/\*

white - returns the colour, white.

\*/

unsigned int white (void);

/\*

black - returns the colour, black.

\*/

unsigned int black (void);

/\*

red - returns the colour, red.

\*/

unsigned int red (void);

/\*

green - returns the colour, green.

\*/

unsigned int green (void);

/\*

blue - returns the colour, blue.

\*/

unsigned int blue (void);

/\*

yellow - returns the colour, yellow.

\*/

unsigned int yellow (void);

/\*

purple - returns the colour, purple.

\*/

unsigned int purple (void);

/\*

gravity - turn on gravity at: g m^2

\*/

void gravity (double g);

/\*

box - place a box in the world at (x0,y0),(x0+i,y0+j)

\*/

unsigned int box (double x0, double y0, double i, double j, unsigned int c);

/\*

poly3 - place a triangle in the world at:

(x0,y0),(x1,y1),(x2,y2)

\*/

unsigned int poly3 (double x0, double y0, double x1, double y1, double x2, double y2, unsigned int c);

/\*

poly4 - place a rectangle in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3)

\*/

unsigned int poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, unsigned int c);

/\*

poly5 - place a pentagon in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3),(x4,y4)

\*/

unsigned int poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, unsigned int c);

/\*

poly6 - place a hexagon in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5)

\*/

unsigned int poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, unsigned int c);

/\*

mass - specify the mass of an object and return the, id.

\*/

unsigned int mass (unsigned int id, double m);

/\*

get\_mass - returns the mass of a circle or polygon object.

\*/

double get\_mass (unsigned int id);

/\*

get\_gravity - returns the gravity of a circle or polygon object.

\*/

double get\_gravity (unsigned int id);

/\*

set\_gravity - sets the per object gravity, g, to a circle or

polygon object.

\*/

void set\_gravity (unsigned int id, double g);

/\*

fix - fix the object to the world.

\*/

unsigned int fix (unsigned int id);

/\*

unfix - unfix the object from the world.

\*/

unsigned int unfix (unsigned int id);

/\*

is\_fixed - returns TRUE if the object, id, is fixed.

\*/

unsigned int is\_fixed (unsigned int id);

/\*

spring - join object, id1, and, id2, with a string of defined

by hooks constant, k, the spring is at rest if it has

length, l. If l < 0 then the game engine considers

the spring to naturally be at rest for the distance

between id1 and id2.

\*/

unsigned int spring (unsigned int id1, unsigned int id2, double k, double d, double l);

/\*

circle - adds a circle to the world. Center

defined by: x0, y0 radius, radius.

\*/

unsigned int circle (double x0, double y0, double radius, unsigned int c);

/\*

velocity - give an object, id, a velocity, vx, vy.

\*/

unsigned int velocity (unsigned int id, double vx, double vy);

/\*

accel - give an object, id, an acceleration, ax, ay.

\*/

unsigned int accel (unsigned int id, double ax, double ay);

/\*

rotate - rotates object with a angular velocity, angle.

\*/

unsigned int rotate (unsigned int id, double angle);

/\*

is\_visible - Gets boolean value for visibility.

\*/

unsigned int is\_visible (unsigned int id);

/\*

visibility - Sets boolean value for visibility

\*/

unsigned int visibility (unsigned int id, unsigned int on);

/\*

fps - set frames per second.

\*/

unsigned int is\_collision (void);

/\*

fps - set frames per second.

\*/

unsigned int is\_frame (void);

/\*

fps - set frames per second.

\*/

unsigned int is\_function (void);

/\*

is\_spring - returns TRUE if the next event is a spring event.

\*/

unsigned int is\_spring (void);

/\*

create\_function\_event - creates a function event at time, t,

in the future. Function id is called

with parameter, param.

\*/

void create\_function\_event (double t, unsigned int id, unsigned int param);

/\*

time\_until - returns the relative time from now until the next event.

\*/

double time\_until (void);

/\*

skip\_until - advances time for, t, units or until the next event is reached.

The amount of time skipped is returned. This function will not

skip past the next event.

\*/

double skip\_until (double t);

/\*

process\_event - advance time to the next event and then

process the event.

\*/

void process\_event (void);

/\*

fps - set frames per second.

\*/

double get\_time (void);

/\*

rm - delete this object from the simulated world.

The same id is returned.

\*/

unsigned int rm (unsigned int id);

/\*

get\_xpos - returns the first point, x, coordinate of object.

\*/

double get\_xpos (unsigned int id);

/\*

get\_ypos - returns the first point, y, coordinate of object.

\*/

double get\_ypos (unsigned int id);

/\*

get\_xvel - returns the X velocity of object.

\*/

double get\_xvel (unsigned int id);

/\*

get\_yvel - returns the Y velocity of object.

\*/

double get\_yvel (unsigned int id);

/\*

get\_xaccel - returns the X accelaration of object.

\*/

double get\_xaccel (unsigned int id);

/\*

get\_yaccel - returns the Y accelaration of object.

\*/

double get\_yaccel (unsigned int id);

/\*

set\_rotate - change the angle of object id to angle.

\*/

void set\_rotate (unsigned int id, double angle);

/\*

put\_xvel - assigns the X velocity of object.

\*/

void put\_xvel (unsigned int id, double r);

/\*

put\_yvel - assigns the Y velocity of object.

\*/

void put\_yvel (unsigned int id, double r);

/\*

put\_xaccel - assigns the X accelaration of object.

\*/

void put\_xaccel (unsigned int id, double r);

/\*

put\_yaccel - assigns the Y accelaration of object.

\*/

void put\_yaccel (unsigned int id, double r);

/\*

set\_colour - sets colour of object, id, to, c.

\*/

void set\_colour (unsigned int id, unsigned int c);

/\*

set\_elasticity - sets elasticity of object, id, to, elasticity.

\*/

void set\_elasticity (unsigned int id, double elasticity);

/\*

get\_elasticity - sets elasticity of object, id, to, elasticity.

\*/

double get\_elasticity (unsigned int id);

/\*

draw\_spring - draw spring, id, using colour, c, and a width, w.

\*/

void draw\_spring (unsigned int id, unsigned int c, double w);

/\*

end\_spring - draw the objects at the end of the spring with

colour, c, when the object comes to rest.

\*/

void end\_spring (unsigned int id, unsigned int c);

/\*

mid\_spring - when the spring reaches its rest point draw

the objects connected by the spring with

colour, c.

\*/

void mid\_spring (unsigned int id, unsigned int c);

/\*

when\_spring - when the spring, id, reaches, length call, func.

\*/

void when\_spring (unsigned int id, double length, unsigned int func);

/\*

apply\_impulse - applies an impulse of magnitude along vector

[x, y] for object, id.

\*/

void apply\_impulse (unsigned int id, double x, double y, double m);

/\*

moving\_towards - returns TRUE if object, id, is moving towards

a point x, y.

\*/

unsigned int moving\_towards (unsigned int id, double x, double y);

/\*

batch - use the batch device to record the output frames.

\*/

void batch (void);

/\*

use\_buffer - use the buffer device to record the output frames.

\*/

void use\_buffer (void);

/\*

empty\_cbuffer - empty the colour buffer.

\*/

void empty\_cbuffer (void);

/\*

empty\_fbuffer - empty the frame buffer.

\*/

void empty\_fbuffer (void);

/\*

use\_time\_delay - should the frame buffer include the time delay command?

\*/

void use\_time\_delay (unsigned int on);

/\*

draw\_collision - turn on drawing of the actual collision frame

and the prediction frame.

\*/

void draw\_collision (unsigned int actual, unsigned int predict);

/\*

set\_collision\_colour - when two objects collide they will both be draw using

colour, c.

\*/

void set\_collision\_colour (unsigned int c);

/\*

dump\_world - dump a list of all objects and their characteristics.

\*/

void dump\_world (void);

/\*

check\_objects - perform a check to make sure that all non fixed

objects have a mass and return TRUE if this is

the case.

\*/

unsigned int check\_objects (void);

/\*

l2h - translate a twoDsim, id, to the pgeid.

\*/

unsigned int l2h (unsigned int id);

/\*

h2l - translate a pgeif, id, to the twoDsim.

\*/

unsigned int h2l (unsigned int id);

/\*

fps - set frames per second.

\*/

void fps (double f);

/\*

trace -

\*/

static unsigned int trace (unsigned int id, char \*name\_, unsigned int \_name\_high);

/\*

Assert -

\*/

static void Assert (unsigned int b);

/\*

init - initialise the modules data structures.

\*/

static void init (void);

/\*

newDef -

\*/

static def newDef (TypeOfDef t, unsigned int d);

/\*

addDef - adds a definition (type, d) into the global list and

returns an index to the definition, id.

\*/

static unsigned int addDef (TypeOfDef type, unsigned int d);

/\*

lookupDef - return the definition of, d, and check its type

is, t.

\*/

static unsigned int lookupDef (TypeOfDef t, unsigned int d);

/\*

check\_range -

\*/

static double check\_range (double r, char \*function\_, unsigned int \_function\_high, char \*param\_, unsigned int \_param\_high);

/\*

nofree - do not free, a.

\*/

static void \* nofree (void \* a);

/\*

trace -

\*/

static unsigned int trace (unsigned int id, char \*name\_, unsigned int \_name\_high)

{

char name[\_name\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (name, name\_, \_name\_high+1);

if (tracing)

{

libc\_printf ((char \*) "pgeif: %s as id=%d\\n", 21, &name, id);

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

Assert -

\*/

static void Assert (unsigned int b)

{

if (! b)

{

libc\_printf ((char \*) "assert failed\\n", 15);

libc\_exit (1);

}

}

/\*

init - initialise the modules data structures.

\*/

static void init (void)

{

listOfDefs = Indexing\_InitIndex (1);

}

/\*

newDef -

\*/

static def newDef (TypeOfDef t, unsigned int d)

{

def f;

Storage\_ALLOCATE ((void \*\*) &f, sizeof (\_T1));

f->type = t;

f->definition = d;

return f;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

addDef - adds a definition (type, d) into the global list and

returns an index to the definition, id.

\*/

static unsigned int addDef (TypeOfDef type, unsigned int d)

{

unsigned int id;

def f;

f = newDef (type, d);

Indexing\_IncludeIndiceIntoIndex (listOfDefs, (void \*) f);

id = Indexing\_HighIndice (listOfDefs);

Assert ((Indexing\_GetIndice (listOfDefs, id)) == f);

if (debugging)

{

libc\_printf ((char \*) "pgeif: map (pgeid %d) onto (twoDsim %d)\\n", 42, id, d);

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

lookupDef - return the definition of, d, and check its type

is, t.

\*/

static unsigned int lookupDef (TypeOfDef t, unsigned int d)

{

def f;

if (debugging)

{

libc\_printf ((char \*) "inside lookupDef (d = %d)\\n", 27, d);

}

if (Indexing\_InBounds (listOfDefs, d))

{

f = Indexing\_GetIndice (listOfDefs, d);

if (debugging)

{

libc\_printf ((char \*) "inside lookupDef (type = %d, definition = %d)\\n", 47, f->type, f->definition);

}

if (t == f->type)

{

return f->definition;

}

else

{

if (debugging)

{

libc\_printf ((char \*) "throwing an exception in lookupDef (1) t = %d, type = %d\\n", 59, t, f->type);

}

throw ( ((unsigned int) (IncorrectType)));

}

}

else

{

if (debugging)

{

libc\_printf ((char \*) "throwing an exception in lookupDef (2)\\n", 40);

}

throw ( ((unsigned int) (IdOutOfBounds)));

}

ReturnException ("../git-pge/m2/pgeif.def", 1, 15);

\_\_builtin\_unreachable ();

}

/\*

check\_range -

\*/

static double check\_range (double r, char \*function\_, unsigned int \_function\_high, char \*param\_, unsigned int \_param\_high)

{

char function[\_function\_high+1];

char param[\_param\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (function, function\_, \_function\_high+1);

memcpy (param, param\_, \_param\_high+1);

if (roots\_nearZero (r))

{

return 0.0;

}

else if (roots\_nearZero (r-1.0))

{

/\* avoid dangling else. \*/

return 1.0;

}

else if ((r > 0.0) && (r < 1.0))

{

/\* avoid dangling else. \*/

return r;

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "%s: parameter value %s is out of range (%g) (using 0.0)\\n", 57, &function, &param, r);

/\* THROW (ORD (ValueOutOfRange)) \*/

return 0.0;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

nofree - do not free, a.

\*/

static void \* nofree (void \* a)

{

/\* do nothing \*/

return a;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

rgb - make a colour object using red, blue and green components.

The colour object is returned.

\*/

unsigned int rgb (double r, double g, double b)

{

Fractions\_Fract rf;

Fractions\_Fract gf;

Fractions\_Fract bf;

rf = Fractions\_putReal (r);

gf = Fractions\_putReal (g);

bf = Fractions\_putReal (b);

return trace (addDef ((TypeOfDef) colour, deviceIf\_defineColour (rf, gf, bf)), (char \*) "colour", 6);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

white - returns the colour, white.

\*/

unsigned int white (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_white ()), (char \*) "white", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

black - returns the colour, black.

\*/

unsigned int black (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_black ()), (char \*) "black", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

red - returns the colour, red.

\*/

unsigned int red (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_red ()), (char \*) "red", 3);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

green - returns the colour, green.

\*/

unsigned int green (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_green ()), (char \*) "green", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

blue - returns the colour, blue.

\*/

unsigned int blue (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_blue ()), (char \*) "blue", 4);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

yellow - returns the colour, yellow.

\*/

unsigned int yellow (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_yellow ()), (char \*) "yellow", 6);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

purple - returns the colour, purple.

\*/

unsigned int purple (void)

{

return trace (addDef ((TypeOfDef) colour, deviceIf\_purple ()), (char \*) "purple", 6);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

gravity - turn on gravity at: g m^2

\*/

void gravity (double g)

{

twoDsim\_gravity (g);

}

/\*

box - place a box in the world at (x0,y0),(x0+i,y0+j)

\*/

unsigned int box (double x0, double y0, double i, double j, unsigned int c)

{

double k;

x0 = check\_range (x0, (char \*) "box", 3, (char \*) "x0", 2);

y0 = check\_range (y0, (char \*) "box", 3, (char \*) "y0", 2);

k = check\_range (x0+i, (char \*) "box", 3, (char \*) "x0+i", 4);

k = check\_range (y0+j, (char \*) "box", 3, (char \*) "y0+j", 4);

return trace (addDef ((TypeOfDef) object, twoDsim\_box (x0, y0, i, j, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c))), (char \*) "box", 3);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly3 - place a triangle in the world at:

(x0,y0),(x1,y1),(x2,y2)

\*/

unsigned int poly3 (double x0, double y0, double x1, double y1, double x2, double y2, unsigned int c)

{

double k;

x0 = check\_range (x0, (char \*) "poly3", 5, (char \*) "x0", 2);

y0 = check\_range (y0, (char \*) "poly3", 5, (char \*) "y0", 2);

x1 = check\_range (x1, (char \*) "poly3", 5, (char \*) "x1", 2);

y1 = check\_range (y1, (char \*) "poly3", 5, (char \*) "y1", 2);

x2 = check\_range (x2, (char \*) "poly3", 5, (char \*) "x2", 2);

y2 = check\_range (y2, (char \*) "poly3", 5, (char \*) "y2", 2);

return trace (addDef ((TypeOfDef) object, twoDsim\_poly3 (x0, y0, x1, y1, x2, y2, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c))), (char \*) "poly3", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly4 - place a rectangle in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3)

\*/

unsigned int poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, unsigned int c)

{

double k;

x0 = check\_range (x0, (char \*) "poly4", 5, (char \*) "x0", 2);

y0 = check\_range (y0, (char \*) "poly4", 5, (char \*) "y0", 2);

x1 = check\_range (x1, (char \*) "poly4", 5, (char \*) "x1", 2);

y1 = check\_range (y1, (char \*) "poly4", 5, (char \*) "y1", 2);

x2 = check\_range (x2, (char \*) "poly4", 5, (char \*) "x2", 2);

y2 = check\_range (y2, (char \*) "poly4", 5, (char \*) "y2", 2);

x3 = check\_range (x3, (char \*) "poly4", 5, (char \*) "x3", 2);

y3 = check\_range (y3, (char \*) "poly4", 5, (char \*) "y3", 2);

return trace (addDef ((TypeOfDef) object, twoDsim\_poly4 (x0, y0, x1, y1, x2, y2, x3, y3, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c))), (char \*) "poly4", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly5 - place a pentagon in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3),(x4,y4)

\*/

unsigned int poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, unsigned int c)

{

double k;

x0 = check\_range (x0, (char \*) "poly5", 5, (char \*) "x0", 2);

y0 = check\_range (y0, (char \*) "poly5", 5, (char \*) "y0", 2);

x1 = check\_range (x1, (char \*) "poly5", 5, (char \*) "x1", 2);

y1 = check\_range (y1, (char \*) "poly5", 5, (char \*) "y1", 2);

x2 = check\_range (x2, (char \*) "poly5", 5, (char \*) "x2", 2);

y2 = check\_range (y2, (char \*) "poly5", 5, (char \*) "y2", 2);

x3 = check\_range (x3, (char \*) "poly5", 5, (char \*) "x3", 2);

y3 = check\_range (y3, (char \*) "poly5", 5, (char \*) "y3", 2);

x4 = check\_range (x4, (char \*) "poly5", 5, (char \*) "x4", 2);

y4 = check\_range (y4, (char \*) "poly5", 5, (char \*) "y4", 2);

return trace (addDef ((TypeOfDef) object, twoDsim\_poly5 (x0, y0, x1, y1, x2, y2, x3, y3, x4, y4, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c))), (char \*) "poly5", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly6 - place a hexagon in the world at:

(x0,y0),(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5)

\*/

unsigned int poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, unsigned int c)

{

double k;

x0 = check\_range (x0, (char \*) "poly6", 5, (char \*) "x0", 2);

y0 = check\_range (y0, (char \*) "poly6", 5, (char \*) "y0", 2);

x1 = check\_range (x1, (char \*) "poly6", 5, (char \*) "x1", 2);

y1 = check\_range (y1, (char \*) "poly6", 5, (char \*) "y1", 2);

x2 = check\_range (x2, (char \*) "poly6", 5, (char \*) "x2", 2);

y2 = check\_range (y2, (char \*) "poly6", 5, (char \*) "y2", 2);

x3 = check\_range (x3, (char \*) "poly6", 5, (char \*) "x3", 2);

y3 = check\_range (y3, (char \*) "poly6", 5, (char \*) "y3", 2);

x4 = check\_range (x4, (char \*) "poly6", 5, (char \*) "x4", 2);

y4 = check\_range (y4, (char \*) "poly6", 5, (char \*) "y4", 2);

x5 = check\_range (x5, (char \*) "poly6", 5, (char \*) "x5", 2);

y5 = check\_range (y5, (char \*) "poly6", 5, (char \*) "y5", 2);

return trace (addDef ((TypeOfDef) object, twoDsim\_poly6 (x0, y0, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c))), (char \*) "poly6", 5);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

mass - specify the mass of an object and return the, id.

\*/

unsigned int mass (unsigned int id, double m)

{

unsigned int ti;

ti = trace (twoDsim\_mass (lookupDef ((TypeOfDef) object, id), m), (char \*) "mass", 4);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_mass - returns the mass of a circle or polygon object.

\*/

double get\_mass (unsigned int id)

{

return twoDsim\_get\_mass (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_gravity - returns the gravity of a circle or polygon object.

\*/

double get\_gravity (unsigned int id)

{

return twoDsim\_get\_gravity (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

set\_gravity - sets the per object gravity, g, to a circle or

polygon object.

\*/

void set\_gravity (unsigned int id, double g)

{

twoDsim\_set\_gravity (lookupDef ((TypeOfDef) object, id), g);

}

/\*

fix - fix the object to the world.

\*/

unsigned int fix (unsigned int id)

{

unsigned int ti;

ti = trace (twoDsim\_fix (lookupDef ((TypeOfDef) object, id)), (char \*) "fix", 3);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

unfix - unfix the object from the world.

\*/

unsigned int unfix (unsigned int id)

{

unsigned int ti;

ti = trace (twoDsim\_unfix (lookupDef ((TypeOfDef) object, id)), (char \*) "unfix", 5);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

is\_fixed - returns TRUE if the object, id, is fixed.

\*/

unsigned int is\_fixed (unsigned int id)

{

return twoDsim\_isFixed (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

spring - join object, id1, and, id2, with a string of defined

by hooks constant, k, the spring is at rest if it has

length, l. If l < 0 then the game engine considers

the spring to naturally be at rest for the distance

between id1 and id2.

\*/

unsigned int spring (unsigned int id1, unsigned int id2, double k, double d, double l)

{

unsigned int ti;

unsigned int id;

libc\_printf ((char \*) "before twoDsim.spring\\n", 23);

ti = twoDsim\_spring (lookupDef ((TypeOfDef) object, id1), lookupDef ((TypeOfDef) object, id2), k, d, l);

libc\_printf ((char \*) "before addDef\\n", 15);

id = addDef ((TypeOfDef) object, ti);

libc\_printf ((char \*) "before lookupDef\\n", 18);

Assert (ti == (lookupDef ((TypeOfDef) object, id)));

libc\_printf ((char \*) "before trace\\n", 14);

return trace (id, (char \*) "spring", 6);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

circle - adds a circle to the world. Center

defined by: x0, y0 radius, radius.

\*/

unsigned int circle (double x0, double y0, double radius, unsigned int c)

{

unsigned int ti;

unsigned int id;

ti = twoDsim\_circle (x0, y0, radius, (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, (unsigned int) c));

id = addDef ((TypeOfDef) object, ti);

Assert (ti == (lookupDef ((TypeOfDef) object, id)));

return trace (id, (char \*) "circle", 6);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

velocity - give an object, id, a velocity, vx, vy.

\*/

unsigned int velocity (unsigned int id, double vx, double vy)

{

unsigned int ti;

if (debugging)

{

libc\_printf ((char \*) "inside velocity (id = %d)\\n", 27, id);

}

ti = trace (twoDsim\_velocity (lookupDef ((TypeOfDef) object, id), vx, vy), (char \*) "velocity", 8);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

accel - give an object, id, an acceleration, ax, ay.

\*/

unsigned int accel (unsigned int id, double ax, double ay)

{

unsigned int ti;

ti = twoDsim\_accel (lookupDef ((TypeOfDef) object, id), ax, ay);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

rotate - rotates object with a angular velocity, angle.

\*/

unsigned int rotate (unsigned int id, double angle)

{

unsigned int ti;

ti = twoDsim\_rotate (lookupDef ((TypeOfDef) object, id), angle);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

is\_visible - Gets boolean value for visibility.

\*/

unsigned int is\_visible (unsigned int id)

{

unsigned int ti;

if(twoDsim\_is\_visible (lookupDef ((TypeOfDef) object, id)))

return 1;

else

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

visibility - Sets boolean value for visibility

\*/

unsigned int visibility (unsigned int id, unsigned int on)

{

unsigned int ti;

ti = twoDsim\_visibility (lookupDef ((TypeOfDef) object, id), on);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fps - set frames per second.

\*/

unsigned int is\_collision (void)

{

return twoDsim\_isCollision ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fps - set frames per second.

\*/

unsigned int is\_frame (void)

{

return twoDsim\_isFrame ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fps - set frames per second.

\*/

unsigned int is\_function (void)

{

return twoDsim\_isFunction ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

is\_spring - returns TRUE if the next event is a spring event.

\*/

unsigned int is\_spring (void)

{

return twoDsim\_isSpring ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

create\_function\_event - creates a function event at time, t,

in the future. Function id is called

with parameter, param.

\*/

void create\_function\_event (double t, unsigned int id, unsigned int param)

{

twoDsim\_createFunctionEvent (t, id, param);

}

/\*

time\_until - returns the relative time from now until the next event.

\*/

double time\_until (void)

{

return twoDsim\_timeUntil ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

skip\_until - advances time for, t, units or until the next event is reached.

The amount of time skipped is returned. This function will not

skip past the next event.

\*/

double skip\_until (double t)

{

return twoDsim\_skipTime (t);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

process\_event - advance time to the next event and then

process the event.

\*/

void process\_event (void)

{

twoDsim\_processEvent ();

}

/\*

fps - set frames per second.

\*/

double get\_time (void)

{

return twoDsim\_getTime ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

rm - delete this object from the simulated world.

The same id is returned.

\*/

unsigned int rm (unsigned int id)

{

unsigned int ti;

ti = twoDsim\_rm (lookupDef ((TypeOfDef) object, id));

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_xpos - returns the first point, x, coordinate of object.

\*/

double get\_xpos (unsigned int id)

{

return check\_range (twoDsim\_get\_xpos (lookupDef ((TypeOfDef) object, id)), (char \*) "get\_xpos", 8, (char \*) "id", 2);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_ypos - returns the first point, y, coordinate of object.

\*/

double get\_ypos (unsigned int id)

{

return check\_range (twoDsim\_get\_ypos (lookupDef ((TypeOfDef) object, id)), (char \*) "get\_ypos", 8, (char \*) "id", 2);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_xvel - returns the X velocity of object.

\*/

double get\_xvel (unsigned int id)

{

return twoDsim\_get\_xvel (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_yvel - returns the Y velocity of object.

\*/

double get\_yvel (unsigned int id)

{

return twoDsim\_get\_yvel (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_xaccel - returns the X accelaration of object.

\*/

double get\_xaccel (unsigned int id)

{

return twoDsim\_get\_xaccel (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_yaccel - returns the Y accelaration of object.

\*/

double get\_yaccel (unsigned int id)

{

return twoDsim\_get\_yaccel (lookupDef ((TypeOfDef) object, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

set\_rotate - change the angle of object id to angle.

\*/

void set\_rotate (unsigned int id, double angle)

{

return twoDsim\_set\_rotate (lookupDef ((TypeOfDef) object, id), angle);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

put\_xvel - assigns the X velocity of object.

\*/

void put\_xvel (unsigned int id, double r)

{

twoDsim\_put\_xvel (lookupDef ((TypeOfDef) object, id), r);

}

/\*

put\_yvel - assigns the Y velocity of object.

\*/

void put\_yvel (unsigned int id, double r)

{

twoDsim\_put\_yvel (lookupDef ((TypeOfDef) object, id), r);

}

/\*

put\_xaccel - assigns the X accelaration of object.

\*/

void put\_xaccel (unsigned int id, double r)

{

twoDsim\_put\_xaccel (lookupDef ((TypeOfDef) object, id), r);

}

/\*

put\_yaccel - assigns the Y accelaration of object.

\*/

void put\_yaccel (unsigned int id, double r)

{

twoDsim\_put\_yaccel (lookupDef ((TypeOfDef) object, id), r);

}

/\*

set\_colour - sets colour of object, id, to, c.

\*/

void set\_colour (unsigned int id, unsigned int c)

{

twoDsim\_set\_colour (lookupDef ((TypeOfDef) object, id), (deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c));

}

/\*

set\_elasticity - sets colour of object, id, to, elasticity.

\*/

void set\_elasticity (unsigned int id, double elasticity)

{

twoDsim\_set\_elasticity (lookupDef ((TypeOfDef) object, id), elasticity);

}

/\*

get\_elasticity - sets colour of object, id, to, elasticity.

\*/

double get\_elasticity (unsigned int id)

{

twoDsim\_get\_elasticity (lookupDef ((TypeOfDef) object, id));

}

/\*

draw\_spring - draw spring, id, using colour, c, and a width, w.

\*/

void draw\_spring (unsigned int id, unsigned int c, double w)

{

twoDsim\_draw\_spring (lookupDef ((TypeOfDef) object, id), lookupDef ((TypeOfDef) colour, c), w);

}

/\*

end\_spring - draw the objects at the end of the spring with

colour, c, when the object comes to rest.

\*/

void end\_spring (unsigned int id, unsigned int c)

{

twoDsim\_end\_spring (lookupDef ((TypeOfDef) object, id), lookupDef ((TypeOfDef) colour, c));

}

/\*

mid\_spring - when the spring reaches its rest point draw

the objects connected by the spring with

colour, c.

\*/

void mid\_spring (unsigned int id, unsigned int c)

{

twoDsim\_mid\_spring (lookupDef ((TypeOfDef) object, id), lookupDef ((TypeOfDef) colour, c));

}

/\*

when\_spring - when the spring, id, reaches, length call, func.

\*/

void when\_spring (unsigned int id, double length, unsigned int func)

{

twoDsim\_when\_spring (lookupDef ((TypeOfDef) object, id), length, func);

}

/\*

apply\_impulse - applies an impulse of magnitude along vector

[x, y] for object, id.

\*/

void apply\_impulse (unsigned int id, double x, double y, double m)

{

twoDsim\_apply\_impulse (lookupDef ((TypeOfDef) object, id), x, y, m);

}

/\*

moving\_towards - returns TRUE if object, id, is moving towards

a point x, y.

\*/

unsigned int moving\_towards (unsigned int id, double x, double y)

{

return twoDsim\_moving\_towards (lookupDef ((TypeOfDef) object, id), x, y);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

batch - use the batch device to record the output frames.

\*/

void batch (void)

{

deviceIf\_useGroff ();

}

/\*

use\_buffer - use the buffer device to record the output frames.

\*/

void use\_buffer (void)

{

double t;

deviceIf\_useBuffer ();

t = time\_until ();

}

/\*

empty\_cbuffer - empty the colour buffer.

\*/

void empty\_cbuffer (void)

{

twoDsim\_emptyCbuffer ();

}

/\*

empty\_fbuffer - empty the frame buffer.

\*/

void empty\_fbuffer (void)

{

twoDsim\_emptyFbuffer ();

}

/\*

use\_time\_delay - should the frame buffer include the time delay command?

\*/

void use\_time\_delay (unsigned int on)

{

twoDsim\_useTimeDelay (on);

}

/\*

draw\_collision - turn on drawing of the actual collision frame

and the prediction frame.

\*/

void draw\_collision (unsigned int actual, unsigned int predict)

{

twoDsim\_drawCollisionFrames (actual, predict);

}

/\*

set\_collision\_colour - when two objects collide they will both be draw using

colour, c.

\*/

void set\_collision\_colour (unsigned int c)

{

twoDsim\_setCollisionColour ((deviceIf\_Colour) lookupDef ((TypeOfDef) colour, c));

}

/\*

dump\_world - dump a list of all objects and their characteristics.

\*/

void dump\_world (void)

{

twoDsim\_dumpWorld ();

}

/\*

check\_objects - perform a check to make sure that all non fixed

objects have a mass and return TRUE if this is

the case.

\*/

unsigned int check\_objects (void)

{

return twoDsim\_checkObjects ();

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

l2h - translate a twoDsim, id, to the pgeid.

\*/

unsigned int l2h (unsigned int id)

{

def d;

unsigned int i;

unsigned int h;

h = Indexing\_HighIndice (listOfDefs);

i = 1;

while (i <= h)

{

d = Indexing\_GetIndice (listOfDefs, i);

if ((d->definition == id) && (d->type == object))

{

return i;

}

i += 1;

}

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

h2l - translate a pgeif, id, to the twoDsim.

\*/

unsigned int h2l (unsigned int id)

{

def d;

d = Indexing\_GetIndice (listOfDefs, id);

return d->definition;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fps - set frames per second.

\*/

void fps (double f)

{

twoDsim\_fps (f);

}

void \_M2\_pgeif\_init (\_\_attribute\_\_((unused)) int argc, \_\_attribute\_\_((unused)) char \*argv[])

{

init ();

}

void \_M2\_pgeif\_finish (\_\_attribute\_\_((unused)) int argc, \_\_attribute\_\_((unused)) char \*argv[])

{

}

**c/Gpgeif.h**

/\* do not edit automatically generated by mc from pgeif. \*/

/\* This file is part of GNU Modula-2.

GNU Modula-2 is free software; you can redistribute it and/or modify it under

the terms of the GNU General Public License as published by the Free

Software Foundation; either version 3, or (at your option) any later

version.

GNU Modula-2 is distributed in the hope that it will be useful, but WITHOUT ANY

WARRANTY; without even the implied warranty of MERCHANTABILITY or

FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License

for more details.

You should have received a copy of the GNU General Public License along

with gm2; see the file COPYING. If not, write to the Free Software

Foundation, 51 Franklin Street, Fifth Floor,

Boston, MA 02110-1301, USA. \*/

#if !defined (\_H)

# define \_H

# ifdef \_\_cplusplus

extern "C" {

# endif

# if !defined (PROC\_D)

# define PROC\_D

typedef void (\*PROC\_t) (void);

typedef struct { PROC\_t proc; } PROC;

# endif

# include "GSYSTEM.h"

# if defined (\_C)

# define EXTERN

# else

# define EXTERN extern

# endif

typedef enum {IncorrectType, IdOutOfBounds, ValueOutOfRange} ExceptionKind;

/\*

rgb - make a colour object using red, blue and green components.

The colour object is returned.

\*/

EXTERN unsigned int rgb (double r, double g, double b);

/\*

white - returns the colour, white.

\*/

EXTERN unsigned int white (void);

/\*

black - returns the colour, black.

\*/

EXTERN unsigned int black (void);

/\*

red - returns the colour, red.

\*/

EXTERN unsigned int red (void);

/\*

green - returns the colour, green.

\*/

EXTERN unsigned int green (void);

/\*

blue - returns the colour, blue.

\*/

EXTERN unsigned int blue (void);

/\*

yellow - returns the colour, yellow.

\*/

EXTERN unsigned int yellow (void);

/\*

purple - returns the colour, purple.

\*/

EXTERN unsigned int purple (void);

/\*

gravity - turn on gravity at: g m^2

\*/

EXTERN void gravity (double g);

/\*

box - place a box in the world at (x0,y0),(x0+i,y0+j)

\*/

EXTERN unsigned int box (double x0, double y0, double i, double j, unsigned int c);

/\*

poly3 - place a triangle in the world at:

(x0,y0), (x1,y1), (x2,y2)

\*/

EXTERN unsigned int poly3 (double x0, double y0, double x1, double y1, double x2, double y2, unsigned int c);

/\*

poly4 - place a rectangle in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3)

\*/

EXTERN unsigned int poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, unsigned int c);

/\*

poly5 - place a pentagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4)

\*/

EXTERN unsigned int poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, unsigned int c);

/\*

poly6 - place a hexagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)

\*/

EXTERN unsigned int poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, unsigned int c);

/\*

mass - specify the mass of an object and return the, id.

\*/

EXTERN unsigned int mass (unsigned int id, double m);

/\*

get\_mass - returns the mass of a circle or polygon object.

\*/

EXTERN double get\_mass (unsigned int id);

/\*

get\_gravity - returns the gravity of a circle or polygon object.

\*/

EXTERN double get\_gravity (unsigned int id);

/\*

set\_gravity - sets the per object gravity, g, to a circle or

polygon object.

\*/

EXTERN void set\_gravity (unsigned int id, double g);

/\*

fix - fix the object to the world.

\*/

EXTERN unsigned int fix (unsigned int id);

/\*

unfix - unfix the object from the world.

\*/

EXTERN unsigned int unfix (unsigned int id);

/\*

is\_fixed - returns TRUE if the object, id, is fixed.

\*/

EXTERN unsigned int is\_fixed (unsigned int id);

/\*

spring - join object, id1, and, id2, with a string of defined

by hooks constant, k, the spring is at rest if it has

length, l. If l < 0 then the game engine considers

the spring to naturally be at rest for the distance

between id1 and id2. The parameter, d, is used to

calculate the damping force.

\*/

EXTERN unsigned int spring (unsigned int id1, unsigned int id2, double k, double d, double l);

/\*

circle - adds a circle to the world. Center

defined by: x0, y0 radius, radius.

\*/

EXTERN unsigned int circle (double x0, double y0, double radius, unsigned int c);

/\*

velocity - give an object, id, a velocity, vx, vy.

\*/

EXTERN unsigned int velocity (unsigned int id, double vx, double vy);

/\*

accel - give an object, id, an acceleration, ax, ay.

\*/

EXTERN unsigned int accel (unsigned int id, double ax, double ay);

/\*

rotate - rotates object with a angular velocity, angle.

\*/

EXTERN unsigned int rotate (unsigned int id, double angle);

/\*

is\_visible - Gets boolean value for visibility.

\*/

EXTERN unsigned int is\_visible (unsigned int id);

/\*

visibility - Sets boolean value for visibility

\*/

EXTERN unsigned int visibility (unsigned int id, unsigned int on);

/\*

is\_collision - returns TRUE if next event is a collision event.

\*/

EXTERN unsigned int is\_collision (void);

/\*

is\_frame - returns TRUE if the next event is a redraw frame event.

\*/

EXTERN unsigned int is\_frame (void);

/\*

is\_function - returns TRUE if the next event is a function event.

\*/

EXTERN unsigned int is\_function (void);

/\*

is\_spring - returns TRUE if the next event is a spring event.

\*/

EXTERN unsigned int is\_spring (void);

/\*

create\_function\_event - creates a function event at time, t,

in the future. Function id is called

with parameter, param.

\*/

EXTERN void create\_function\_event (double t, unsigned int id, unsigned int param);

/\*

time\_until - returns the relative time from now until the next event.

\*/

EXTERN double time\_until (void);

/\*

skip\_until - advances time for, t, units or until the next event is reached.

\*/

EXTERN double skip\_until (double t);

/\*

process\_event - advance time to the next event and then

process the event.

\*/

EXTERN void process\_event (void);

/\*

get\_time - return the current simulation time.

\*/

EXTERN double get\_time (void);

/\*

rm - delete this object from the simulated world.

The same id is returned.

\*/

EXTERN unsigned int rm (unsigned int id);

/\*

get\_xpos - returns the first point, x, coordinate of object.

\*/

EXTERN double get\_xpos (unsigned int id);

/\*

get\_ypos - returns the first point, y, coordinate of object.

\*/

EXTERN double get\_ypos (unsigned int id);

/\*

get\_xvel - returns the X velocity of object.

\*/

EXTERN double get\_xvel (unsigned int id);

/\*

get\_yvel - returns the Y velocity of object.

\*/

EXTERN double get\_yvel (unsigned int id);

/\*

get\_xaccel - returns the X acceleration of object.

\*/

EXTERN double get\_xaccel (unsigned int id);

/\*

get\_yaccel - returns the Y acceleration of object.

\*/

EXTERN double get\_yaccel (unsigned int id);

/\*

put\_xvel - assigns the X velocity of object.

\*/

EXTERN void put\_xvel (unsigned int id, double r);

/\*

put\_yvel - assigns the Y velocity of object.

\*/

EXTERN void put\_yvel (unsigned int id, double r);

/\*

put\_xaccel - assigns the X acceleration of object.

\*/

EXTERN void put\_xaccel (unsigned int id, double r);

/\*

put\_yaccel - assigns the Y acceleration of object.

\*/

EXTERN void put\_yaccel (unsigned int id, double r);

/\*

set\_rotate - change the angle of object id to angle.

\*/

EXTERN void set\_rotate (unsigned int id, double angle);

/\*

set\_colour - sets colour of object, id, to, c.

\*/

EXTERN void set\_colour (unsigned int id, unsigned int c);

/\*

set\_elasticity - sets colour of object, id, to, c.

\*/

EXTERN void set\_elasticity (unsigned int id, double elasticity);

/\*

get\_elasticity - sets colour of object, id, to, c.

\*/

EXTERN double get\_elasticity (unsigned int id);

/\*

draw\_spring - draw a spring, id, using colour, c, with a width, w.

\*/

EXTERN void draw\_spring (unsigned int id, unsigned int c, double w);

/\*

end\_spring - draw the objects at the end of the spring with

colour, c, when the object comes to rest.

\*/

EXTERN void end\_spring (unsigned int id, unsigned int c);

/\*

mid\_spring - when the spring reaches its rest point draw

the objects connected by the spring with

colour, c.

\*/

EXTERN void mid\_spring (unsigned int id, unsigned int c);

/\*

when\_spring - when the spring, id, reaches, length call, func.

\*/

EXTERN void when\_spring (unsigned int id, double length, unsigned int func);

/\*

apply\_impulse - applies an impulse of magnitude along vector

[x, y] for object, id.

\*/

EXTERN void apply\_impulse (unsigned int id, double x, double y, double m);

/\*

moving\_towards - returns TRUE if object, id, is moving towards

a point x, y.

\*/

EXTERN unsigned int moving\_towards (unsigned int id, double x, double y);

/\*

batch - use the batch device to record the output frames.

\*/

EXTERN void batch (void);

/\*

use\_buffer - use the buffer device to record the output frames.

\*/

EXTERN void use\_buffer (void);

/\*

empty\_cbuffer - empty the colour buffer.

\*/

EXTERN void empty\_cbuffer (void);

/\*

empty\_fbuffer - empty the frame buffer.

\*/

EXTERN void empty\_fbuffer (void);

EXTERN void use\_time\_delay (unsigned int on);

/\*

draw\_collision - turn on drawing of the actual collision frame

and the prediction frame.

\*/

EXTERN void draw\_collision (unsigned int actual, unsigned int predict);

/\*

set\_collision\_colour - when two objects collide they will both be draw using

colour, c.

\*/

EXTERN void set\_collision\_colour (unsigned int c);

/\*

dump\_world - dump a list of all objects and their characteristics.

\*/

EXTERN void dump\_world (void);

/\*

check\_objects - perform a check to make sure that all non fixed

objects have a mass and returns TRUE if this is the

case.

\*/

EXTERN unsigned int check\_objects (void);

/\*

l2h - translate a twoDsim, id, to the pgeid.

\*/

EXTERN unsigned int l2h (unsigned int id);

/\*

h2l - translate a pgeif, id, to the twoDsim.

\*/

EXTERN unsigned int h2l (unsigned int id);

/\*

fps - set frames per second.

\*/

EXTERN void fps (double f);

# ifdef \_\_cplusplus

}

# endif

# undef EXTERN

#endif

**python/twoDsim.py**

# This file was automatically generated by SWIG (http://www.swig.org).

# Version 2.0.7

#

# Do not make changes to this file unless you know what you are doing--modify

# the SWIG interface file instead.

from sys import version\_info

if version\_info >= (2,6,0):

def swig\_import\_helper():

from os.path import dirname

import imp

fp = None

try:

fp, pathname, description = imp.find\_module('\_twoDsim', [dirname(\_\_file\_\_)])

except ImportError:

import \_twoDsim

return \_twoDsim

if fp is not None:

try:

\_mod = imp.load\_module('\_twoDsim', fp, pathname, description)

finally:

fp.close()

return \_mod

\_twoDsim = swig\_import\_helper()

del swig\_import\_helper

else:

import \_twoDsim

del version\_info

try:

\_swig\_property = property

except NameError:

pass # Python < 2.2 doesn't have 'property'.

def \_swig\_setattr\_nondynamic(self,class\_type,name,value,static=1):

if (name == "thisown"): return self.this.own(value)

if (name == "this"):

if type(value).\_\_name\_\_ == 'SwigPyObject':

self.\_\_dict\_\_[name] = value

return

method = class\_type.\_\_swig\_setmethods\_\_.get(name,None)

if method: return method(self,value)

if (not static):

self.\_\_dict\_\_[name] = value

else:

raise AttributeError("You cannot add attributes to %s" % self)

def \_swig\_setattr(self,class\_type,name,value):

return \_swig\_setattr\_nondynamic(self,class\_type,name,value,0)

def \_swig\_getattr(self,class\_type,name):

if (name == "thisown"): return self.this.own()

method = class\_type.\_\_swig\_getmethods\_\_.get(name,None)

if method: return method(self)

raise AttributeError(name)

def \_swig\_repr(self):

try: strthis = "proxy of " + self.this.\_\_repr\_\_()

except: strthis = ""

return "<%s.%s; %s >" % (self.\_\_class\_\_.\_\_module\_\_, self.\_\_class\_\_.\_\_name\_\_, strthis,)

try:

\_object = object

\_newclass = 1

except AttributeError:

class \_object : pass

\_newclass = 0

def twoDsim\_skipFor(\*args):

return \_twoDsim.twoDsim\_skipFor(\*args)

twoDsim\_skipFor = \_twoDsim.twoDsim\_skipFor

def twoDsim\_simulateFor(\*args):

return \_twoDsim.twoDsim\_simulateFor(\*args)

twoDsim\_simulateFor = \_twoDsim.twoDsim\_simulateFor

def twoDsim\_replayRate(\*args):

return \_twoDsim.twoDsim\_replayRate(\*args)

twoDsim\_replayRate = \_twoDsim.twoDsim\_replayRate

def twoDsim\_fps(\*args):

return \_twoDsim.twoDsim\_fps(\*args)

twoDsim\_fps = \_twoDsim.twoDsim\_fps

def twoDsim\_rotate(\*args):

return \_twoDsim.twoDsim\_rotate(\*args)

twoDsim\_rotate = \_twoDsim.twoDsim\_rotate

def twoDsim\_accel(\*args):

return \_twoDsim.twoDsim\_accel(\*args)

twoDsim\_accel = \_twoDsim.twoDsim\_accel

def twoDsim\_velocity(\*args):

return \_twoDsim.twoDsim\_velocity(\*args)

twoDsim\_velocity = \_twoDsim.twoDsim\_velocity

def twoDsim\_pivot(\*args):

return \_twoDsim.twoDsim\_pivot(\*args)

twoDsim\_pivot = \_twoDsim.twoDsim\_pivot

def twoDsim\_circle(\*args):

return \_twoDsim.twoDsim\_circle(\*args)

twoDsim\_circle = \_twoDsim.twoDsim\_circle

def twoDsim\_fix(\*args):

return \_twoDsim.twoDsim\_fix(\*args)

twoDsim\_fix = \_twoDsim.twoDsim\_fix

def twoDsim\_mass(\*args):

return \_twoDsim.twoDsim\_mass(\*args)

twoDsim\_mass = \_twoDsim.twoDsim\_mass

def twoDsim\_poly6(\*args):

return \_twoDsim.twoDsim\_poly6(\*args)

twoDsim\_poly6 = \_twoDsim.twoDsim\_poly6

def twoDsim\_poly5(\*args):

return \_twoDsim.twoDsim\_poly5(\*args)

twoDsim\_poly5 = \_twoDsim.twoDsim\_poly5

def twoDsim\_poly4(\*args):

return \_twoDsim.twoDsim\_poly4(\*args)

twoDsim\_poly4 = \_twoDsim.twoDsim\_poly4

def twoDsim\_poly3(\*args):

return \_twoDsim.twoDsim\_poly3(\*args)

twoDsim\_poly3 = \_twoDsim.twoDsim\_poly3

def twoDsim\_box(\*args):

return \_twoDsim.twoDsim\_box(\*args)

twoDsim\_box = \_twoDsim.twoDsim\_box

def twoDsim\_gravity(\*args):

return \_twoDsim.twoDsim\_gravity(\*args)

twoDsim\_gravity = \_twoDsim.twoDsim\_gravity

# This file is compatible with both classic and new-style classes.

**c/twoDsim.c**

/\* do not edit automatically generated by mc from twoDsim. \*/

/\* This file is part of GNU Modula-2.

GNU Modula-2 is free software; you can redistribute it and/or modify it under

the terms of the GNU General Public License as published by the Free

Software Foundation; either version 3, or (at your option) any later

version.

GNU Modula-2 is distributed in the hope that it will be useful, but WITHOUT ANY

WARRANTY; without even the implied warranty of MERCHANTABILITY or

FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License

for more details.

You should have received a copy of the GNU General Public License along

with gm2; see the file COPYING. If not, write to the Free Software

Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA. \*/

# if !defined (PROC\_D)

# define PROC\_D

typedef void (\*PROC\_t) (void);

typedef struct { PROC\_t proc; } PROC;

# endif

# if !defined (TRUE)

# define TRUE (1==1)

# endif

# if !defined (FALSE)

# define FALSE (1==0)

# endif

#include <stddef.h>

#include <string.h>

#include <limits.h>

# include "GStorage.h"

# include "Gmcrts.h"

#define \_twoDsim\_H

#define \_twoDsim\_C

# include "GSYSTEM.h"

# include "GStorage.h"

# include "GIndexing.h"

# include "Glibc.h"

# include "GdeviceIf.h"

# include "Glibm.h"

# include "Groots.h"

# include "GFractions.h"

# include "GPoints.h"

# include "GGC.h"

# include "Gcoord.h"

# include "Gpolar.h"

# include "Ghistory.h"

# include "Gdelay.h"

# include "GMathLib0.h"

# include "GIOChan.h"

# include "GChanConsts.h"

# include "GNetworkOrder.h"

# include "GStrLib.h"

# include "Ginterpen.h"

# include "Gsegment.h"

# include "GMemStream.h"

# include "Ggdbif.h"

# include "GM2RTS.h"

# define MaxPolygonPoints 6

# define DefaultFramesPerSecond 100.0

# define Debugging FALSE

# define DebugTrace FALSE

# define BufferedTime 0.1

# define InactiveTime 1.0

# define Elasticity 0.98

# define ElasticitySpring 0.9

# define FrameSprings TRUE

# define PolygonDebugging FALSE

# define BroadphaseDebugging FALSE

typedef struct descP\_p descP;

typedef struct cDesc\_r cDesc;

typedef struct cpDesc\_r cpDesc;

typedef struct ppDesc\_r ppDesc;

typedef struct fcDesc\_r fcDesc;

typedef struct spDesc\_r spDesc;

typedef struct Spring\_r Spring;

typedef struct Circle\_r Circle;

typedef struct Polygon\_r Polygon;

typedef struct eventProc\_p eventProc;

typedef struct \_T1\_r \_T1;

typedef \_T1 \*eventDesc;

typedef struct \_T2\_r \_T2;

typedef \_T2 \*Object;

typedef struct \_T3\_a \_T3;

typedef struct \_T4\_r \_T4;

typedef \_T4 \*eventQueue;

typedef struct \_T5\_r \_T5;

typedef \_T5 \*broadphase;

typedef enum {polygonOb, circleOb, springOb} ObjectType;

typedef enum {frameKind, functionKind, collisionKind, springKind} eventKind;

typedef enum {frameEvent, circlesEvent, circlePolygonEvent, polygonPolygonEvent, functionEvent, springEvent} eventType;

# define thickness 0.01

typedef eventDesc (\*descP\_t) (eventDesc, unsigned int, unsigned int, unsigned int, unsigned int, history\_whereHit, history\_whereHit, coord\_Coord);

struct descP\_p { descP\_t proc; };

struct cDesc\_r {

coord\_Coord cPoint;

unsigned int cid1;

unsigned int cid2;

};

struct cpDesc\_r {

coord\_Coord cPoint;

unsigned int pid;

unsigned int cid;

history\_whereHit wpid;

unsigned int lineNo;

unsigned int pointNo;

};

struct ppDesc\_r {

coord\_Coord cPoint;

unsigned int pid1;

unsigned int pid2;

history\_whereHit wpid1;

history\_whereHit wpid2;

unsigned int lineCorner1;

unsigned int lineCorner2;

};

struct fcDesc\_r {

unsigned int id;

unsigned int param;

};

struct spDesc\_r {

unsigned int id;

history\_springPoint type;

};

struct Spring\_r {

unsigned int id1;

unsigned int id2;

coord\_Coord f1;

coord\_Coord f2;

coord\_Coord a1;

coord\_Coord a2;

double k;

double d;

double l0;

double cbl;

double l1;

double width;

unsigned int drawColour;

unsigned int endColour;

unsigned int midColour;

unsigned int draw;

unsigned int drawEnd;

unsigned int drawMid;

unsigned int hasCallBackLength;

unsigned int func;

};

struct Circle\_r {

coord\_Coord pos;

double r;

double mass;

deviceIf\_Colour col;

};

typedef void (\*eventProc\_t) (eventQueue);

struct eventProc\_p { eventProc\_t proc; };

struct \_T3\_a { polar\_Polar array[MaxPolygonPoints+1]; };

struct \_T5\_r {

unsigned int o0;

unsigned int o1;

broadphase next;

};

struct Polygon\_r {

unsigned int nPoints;

\_T3 points;

double mass;

deviceIf\_Colour col;

coord\_Coord oldcOfG;

coord\_Coord cOfG;

};

struct \_T1\_r {

eventType etype; /\* case tag \*/

union {

cDesc cc;

cpDesc cp;

ppDesc pp;

fcDesc fc;

spDesc sp;

};

eventDesc next;

};

struct \_T4\_r {

eventKind kind;

double time\_;

eventProc p;

eventDesc ePtr;

eventQueue next;

};

struct \_T2\_r {

unsigned int id;

unsigned int deleted;

unsigned int fixed;

unsigned int stationary;

unsigned int visible;

double gravity;

double elasticity;

coord\_Coord saccel;

coord\_Coord forceVec;

double vx;

double vy;

double ax;

double ay;

double ke;

double pe;

double inertia;

double angleOrientation;

double angularVelocity;

double angularMomentum;

unsigned int interpen;

ObjectType object; /\* case tag \*/

union {

Polygon p;

Circle c;

Spring s;

};

};

static Indexing\_Index objects;

static unsigned int maxId;

static double lastDrawTime;

static double lastUpdateTime;

static double currentTime;

static double replayPerSecond;

static double framesPerSecond;

static double simulatedGravity;

static eventQueue eventQ;

static eventQueue freeEvents;

static eventDesc freeDesc;

static unsigned int trace;

static unsigned int framePolygons;

static unsigned int writeTimeDelay;

static unsigned int drawPrediction;

static unsigned int drawCollisionFrame;

static unsigned int haveSpringColour;

static unsigned int haveCollisionColour;

static deviceIf\_Colour springColour;

static deviceIf\_Colour collisionColour;

static void \* bufferStart;

static unsigned int bufferLength;

static unsigned int bufferUsed;

static unsigned int fileOpened;

static IOChan\_ChanId file;

static unsigned int noOfCulledCollisions;

static unsigned int startedRunning;

static broadphase freeBroadphase;

/\*

gravity - turn on gravity at: g m^2

\*/

void twoDsim\_gravity (double g);

/\*

box - place a box in the world at (x0,y0), (x0+i,y0),

(x0+i, y0+j), (x0, y0+j).

\*/

unsigned int twoDsim\_box (double x0, double y0, double i, double j, deviceIf\_Colour colour);

/\*

poly3 - place a triangle in the world at:

(x0,y0), (x1,y1), (x2,y2)

\*/

unsigned int twoDsim\_poly3 (double x0, double y0, double x1, double y1, double x2, double y2, deviceIf\_Colour colour);

/\*

poly4 - place a quadrangle in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3) the points must be in order,

either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, deviceIf\_Colour colour);

/\*

poly5 - place a pentagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4)

the points must be in order, either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, deviceIf\_Colour colour);

/\*

poly6 - place a hexagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)

the points must be in order, either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, deviceIf\_Colour colour);

/\*

get\_xpos - returns the x coordinate of the center of gravity of object, id.

\*/

double twoDsim\_get\_xpos (unsigned int id);

/\*

get\_ypos - returns the first point, y, coordinate of object.

\*/

double twoDsim\_get\_ypos (unsigned int id);

/\*

get\_xvel - returns the x velocity of object.

\*/

double twoDsim\_get\_xvel (unsigned int id);

/\*

get\_yvel - returns the y velocity of object.

\*/

double twoDsim\_get\_yvel (unsigned int id);

/\*

get\_xaccel - returns the x acceleration of object.

\*/

double twoDsim\_get\_xaccel (unsigned int id);

/\*

get\_yaccel - returns the y acceleration of object.

\*/

double twoDsim\_get\_yaccel (unsigned int id);

/\*

put\_xvel - assigns the x velocity of object.

\*/

void twoDsim\_put\_xvel (unsigned int id, double r);

/\*

put\_yvel - assigns the y velocity of object.

\*/

void twoDsim\_put\_yvel (unsigned int id, double r);

/\*

put\_xaccel - assigns the x acceleration of object.

\*/

void twoDsim\_put\_xaccel (unsigned int id, double r);

/\*

put\_yaccel - assigns the y acceleration of object.

\*/

void twoDsim\_put\_yaccel (unsigned int id, double r);

/\*

apply\_impulse - apply an impulse to object, id,

along the vector [x, y] with magnitude, m.

Nothing happens if the object is fixed.

Currently only circles can have impulses

applied.

\*/

void twoDsim\_apply\_impulse (unsigned int id, double x, double y, double m);

/\*

moving\_towards - returns TRUE if object, id, is moving towards

a point x, y.

\*/

unsigned int twoDsim\_moving\_towards (unsigned int id, double x, double y);

/\*

set\_colour - set the colour of object, id, to colour.

id must be a box or circle.

\*/

void twoDsim\_set\_colour (unsigned int id, deviceIf\_Colour colour);

/\*

set\_elasticity - set the elasticity of object, id, to elasticity.

id must be a box or circle.

\*/

void set\_elasticity (unsigned int id, double elasticity);

/\*

get\_elasticity - set the elasticity of object, id, to elasticity.

id must be a box or circle.

\*/

double get\_elasticity (unsigned int id);

/\*

set\_gravity - set the gravity of object, id, to, g.

id must be a box or circle.

\*/

void twoDsim\_set\_gravity (unsigned int id, double g);

/\*

get\_gravity - return the gravity of object, id.

id must be a box or circle.

\*/

double twoDsim\_get\_gravity (unsigned int id);

/\*

get\_mass - returns the mass of object, id.

\*/

double twoDsim\_get\_mass (unsigned int id);

/\*

mass - specify the mass of an object and return the, id.

Only polygon (and box) and circle objects may have

a mass.

\*/

unsigned int twoDsim\_mass (unsigned int id, double m);

/\*

fix - fix the object to the world.

\*/

unsigned int twoDsim\_fix (unsigned int id);

/\*

unfix - unfix the object from the world.

\*/

unsigned int twoDsim\_unfix (unsigned int id);

/\*

spring - join object, id1, and, id2, with a string of defined

by hooks constant, k, the spring is at rest if it has

length, l. If l < 0 then the game engine considers

the spring to naturally be at rest for the distance

between id1 and id2. The parameter, d, is used to

calculate the damping force.

\*/

unsigned int twoDsim\_spring (unsigned int id1, unsigned int id2, double k, double d, double l);

/\*

draw\_spring - draw spring, id, using colour, c, and a width, w.

\*/

void twoDsim\_draw\_spring (unsigned int id, unsigned int c, double w);

/\*

end\_spring - draw the spring using colour, c, when it reaches the end.

\*/

void twoDsim\_end\_spring (unsigned int id, unsigned int c);

/\*

mid\_spring - when the string reaches its rest point draw the objects

connected.

\*/

void twoDsim\_mid\_spring (unsigned int id, unsigned int c);

/\*

when\_spring - when the spring, id, reaches, length call, func.

\*/

void twoDsim\_when\_spring (unsigned int id, double length, unsigned int func);

/\*

circle - adds a circle to the world. Center

defined by: x0, y0 radius, r.

\*/

unsigned int twoDsim\_circle (double x0, double y0, double radius, deviceIf\_Colour colour);

/\*

velocity - give an object, id, a velocity, vx, vy.

\*/

unsigned int twoDsim\_velocity (unsigned int id, double vx, double vy);

/\*

accel - give an object, id, an acceleration, ax, ay.

\*/

unsigned int twoDsim\_accel (unsigned int id, double ax, double ay);

/\*

rotate - gives object, id, an initial orientation.

\*/

unsigned int twoDsim\_rotate (unsigned int id, double angle);

/\*

is\_visible - Gets boolean value for visibility.

\*/

unsigned int twoDsim\_is\_visible (unsigned int id);

/\*

visibility - Sets boolean value for visibility

\*/

unsigned int twoDsim\_visibility (unsigned int id, unsigned int on);

/\*

rvel - gives object, id, an angular velocity, angle.

\*/

unsigned int twoDsim\_rvel (unsigned int id, double angle);

/\*

fps - set frames per second.

\*/

void twoDsim\_fps (double f);

/\*

replayRate - set frames per second during replay.

\*/

void twoDsim\_replayRate (double f);

/\*

simulateFor - render for, t, seconds.

\*/

void twoDsim\_simulateFor (double t);

/\*

getTime - returns the current time in the simulation.

\*/

double twoDsim\_getTime (void);

/\*

skipTime - attempts to skip, t, seconds. It returns the amount

of time actually skipped. This function will not skip

past the next event.

\*/

double twoDsim\_skipTime (double t);

/\*

timeUntil - returns the relative time from now until the next event.

\*/

double twoDsim\_timeUntil (void);

/\*

processEvent - skips any outstanding time and processes the next event.

Time is adjusted to the time of the next event.

\*/

void twoDsim\_processEvent (void);

/\*

isCollision - returns TRUE if the next event is a collision event.

\*/

unsigned int twoDsim\_isCollision (void);

/\*

isFrame - returns TRUE if the next event is a frame event.

\*/

unsigned int twoDsim\_isFrame (void);

/\*

isFunction - returns TRUE if the next event is a function event.

\*/

unsigned int twoDsim\_isFunction (void);

/\*

isSpring - returns TRUE if the next event is a spring event.

\*/

unsigned int twoDsim\_isSpring (void);

/\*

isFixed - returns TRUE if object, id, is fixed.

\*/

unsigned int twoDsim\_isFixed (unsigned int id);

/\*

createFunctionEvent - creates a function event at time, t,

in the future.

\*/

void twoDsim\_createFunctionEvent (double t, unsigned int id, unsigned int param);

/\*

rm - delete this object from the simulated world.

The same id is returned.

\*/

unsigned int twoDsim\_rm (unsigned int id);

/\*

getEventBuffer - collects the event buffer limits in the following parameters.

\*/

void twoDsim\_getEventBuffer (void \* \*start, unsigned int \*length, unsigned int \*used);

/\*

buildFrame - populate the frame buffer contents with the world at the current time.

\*/

void twoDsim\_buildFrame (void);

/\*

emptyCbuffer - empty the colour buffer.

\*/

void twoDsim\_emptyCbuffer (void);

/\*

emptyFbuffer - empty the frame buffer.

\*/

void twoDsim\_emptyFbuffer (void);

/\*

useTimeDelay - should the frame buffer include the time delay command?

\*/

void twoDsim\_useTimeDelay (unsigned int on);

/\*

drawCollisionFrames - turn the drawing of collision frames on or off.

actual: determines whether an extra frame is generated

at the time of actual collision.

predict: draws a frame predicting the next collision.

It will show the points predicted to collide.

\*/

void twoDsim\_drawCollisionFrames (unsigned int actual, unsigned int predict);

/\*

setCollisionColour - assigns, c, as the colour for objects colliding.

\*/

void twoDsim\_setCollisionColour (deviceIf\_Colour c);

/\*

dumpWorld - dump a list of all objects and their characteristics.

\*/

void twoDsim\_dumpWorld (void);

/\*

checkObjects - perform a check to make sure that all non fixed objects have a mass.

\*/

unsigned int twoDsim\_checkObjects (void);

/\*

Assert -

\*/

static void Assert (unsigned int b, unsigned int line);

/\*

AssertR -

\*/

static void AssertR (double a, double b);

/\*

AssertRFail -

\*/

static void AssertRFail (double a, double b);

/\*

AssertRDebug -

\*/

static void AssertRDebug (double a, double b, char \*message\_, unsigned int \_message\_high);

/\*

dumpSpring -

\*/

static void dumpSpring (Object o);

/\*

dumpCircle -

\*/

static void dumpCircle (Object o);

/\*

dumpPolygon -

\*/

static void dumpPolygon (Object o);

/\*

checkDeleted -

\*/

static void checkDeleted (Object o);

/\*

dumpObject -

\*/

static void dumpObject (Object o);

/\*

safeCoord - ensures that 0.0 <= r <= 1.0.

\*/

static double safeCoord (double r);

/\*

c2p - returns a Point given a Coord.

\*/

static Points\_Point c2p (coord\_Coord c);

/\*

newObject - creates an object of, type, and returns its, id.

\*/

static unsigned int newObject (ObjectType type);

/\*

circle\_moving\_towards - return TRUE if object, optr, is moving towards

point, x, y. The object may still miss point x, y

but it is moving closer to this position.

\*/

static unsigned int circle\_moving\_towards (Object optr, double x, double y);

/\*

apply\_impulse\_to\_circle - apply an impulse to moving circle, movable,

along the vector [x, y] with magnitude, m.

\*/

static void apply\_impulse\_to\_circle (Object movable, double x, double y, double m);

/\*

objectExists - returns TRUE if object, o, has not been deleted.

\*/

static unsigned int objectExists (Object o);

/\*

objectIdExists - returns TRUE if object, id, has not been deleted.

\*/

static unsigned int objectIdExists (unsigned int id);

/\*

doCheckInterpenCircleCircle -

\*/

static unsigned int doCheckInterpenCircleCircle (Object fixed, Object movable);

/\*

doCheckInterpenCircleCircleMoving -

\*/

static unsigned int doCheckInterpenCircleCircleMoving (Object c1, Object c2);

/\*

distanceLinePoint - c is a point. p1->p2 is the line in question.

p3 is assigned to the closest point on the line

to the point, c. d is the distance from c to p3.

TRUE is returned if the point, c, lies above or

below the line once the line is rotated onto the x

axis. (The point, c, would also rotated to solve

this question).

\*/

static unsigned int distanceLinePoint (coord\_Coord c, coord\_Coord p1, coord\_Coord p2, coord\_Coord \*p3, double \*d);

/\*

checkLimits -

\*/

static coord\_Coord checkLimits (coord\_Coord c, double r);

/\*

doCheckInterpenCirclePolygon -

\*/

static unsigned int doCheckInterpenCirclePolygon (Object iptr, Object jptr);

/\*

doCheckInterpenPolygon -

\*/

static unsigned int doCheckInterpenPolygon (Object iptr, Object jptr);

/\*

doCheckInterpenCircle -

\*/

static unsigned int doCheckInterpenCircle (Object iptr, Object jptr);

/\*

initInterpen -

\*/

static void initInterpen (void);

/\*

max - return the maximum of a and b.

\*/

static unsigned int max (unsigned int a, unsigned int b);

/\*

checkMicroInterpenCircle -

\*/

static void checkMicroInterpenCircle (void);

/\*

checkMicroInterpenPolygon -

\*/

static void checkMicroInterpenPolygon (void);

/\*

checkMicroInterpen - this performs micro collision analysis, it detects interpenetration

between objects and separates the objects, without much force.

This is called if we are using collision prediction and this will solve

rounding errors which might otherwise allow objects to fall into each other.

It will keep moving objects apart from each other and keep going in cycles

(for a limited number of cycles).

\*/

static void checkMicroInterpen (void);

/\*

resetStationary -

\*/

static void resetStationary (void);

/\*

getInterCircle - return the interCircle data structure filled in from circle0.

\*/

static interpen\_interCircle getInterCircle (Object circle0);

/\*

checkFrameInterpenCircleCircle -

\*/

static void checkFrameInterpenCircleCircle (Object circle0, Object circle1);

/\*

checkFrameInterpenCirclePolygon -

\*/

static void checkFrameInterpenCirclePolygon (Object circle0, Object polygon0);

/\*

dumpCoord -

\*/

static void dumpCoord (coord\_Coord p);

/\*

dumpCollision -

\*/

static void dumpCollision (unsigned int pid0, unsigned int pid1, unsigned int l0, unsigned int l1, history\_whereHit at0, history\_whereHit at1, coord\_Coord p);

/\*

debugDelay -

\*/

static void debugDelay (char \*message\_, unsigned int \_message\_high);

/\*

restoreOldCofG -

\*/

static void restoreOldCofG (Object poly);

/\*

checkFrameInterpenPolygonPolygon - checks every line segment of polygon0 vs polygon1

and registers a collision event at the current time

if these segments intersect.

\*/

static void checkFrameInterpenPolygonPolygon (Object polygon0, Object polygon1);

/\*

checkFrameInterpenObjects -

\*/

static void checkFrameInterpenObjects (unsigned int i, unsigned int j);

/\*

displayBroadphase -

\*/

static void displayBroadphase (broadphase b);

/\*

dumpBroadphase -

\*/

static void dumpBroadphase (broadphase list);

/\*

optBroadphase - optimise the broadphase. (--fixme--) complete me.

\*/

static broadphase optBroadphase (broadphase list);

/\*

checkFrameInterpen - at this point the engine is running in frame based mode

as we have at least one polygon moving and rotating.

We need to check for interpenetration of objects, move them apart

and add a collision event for each pair of interpenetrating objects.

\*/

static void checkFrameInterpen (void);

/\*

getCofG - returns the CofG of an object.

\*/

static coord\_Coord getCofG (unsigned int id);

/\*

isCircle - return TRUE if object, id, is a circle.

\*/

static unsigned int isCircle (unsigned int id);

/\*

isPolygon - return TRUE if object, id, is a polygon.

\*/

static unsigned int isPolygon (unsigned int id);

/\*

isSpringObject - return TRUE if object, id, is a spring.

\*/

static unsigned int isSpringObject (unsigned int id);

/\*

calcSpringFixed - calculate the forces on, moving object which is attached to, fixed.

Given spring properties of, k, and, l0.

\*/

static void calcSpringFixed (double k, double d, double l0, double l1, unsigned int spr, unsigned int fixed, unsigned int moving);

/\*

doCalcSpringFixed - calculate the forces on, moving object which is attached to, fixed.

Given spring properties of, k, and, l0.

\*/

static void doCalcSpringFixed (double k, double d, double l0, double l1, unsigned int spr, unsigned int fixed, unsigned int moving);

/\*

calcSpringMoving - calculate the forces on, moving objects o1 and o2 attached to

spring, spr.

The spring has properties of, k, d, l0 and l1.

\*/

static void calcSpringMoving (double k, double d, double l0, double l1, unsigned int spr, unsigned int o1, unsigned int o2);

/\*

doCalcSpringMoving - calculate the forces on, moving objects o1 and o2 attached to

spring, spr.

The spring has properties of, k, d, l0 and l1.

\*/

static void doCalcSpringMoving (double k, double d, double l0, double l1, unsigned int spr, unsigned int o1, unsigned int o2);

/\*

doCalcSpringForce -

\*/

static void doCalcSpringForce (unsigned int id, Object idp);

/\*

calcSpringForce - calculate the forces a spring, id, has on its components.

\*/

static void calcSpringForce (unsigned int id);

/\*

zeroForceEnergy - assign force vector, potential energy and kinetic energy

to zero for all objects

\*/

static void zeroForceEnergy (void);

/\*

applyDrag - apply drag to object, id, which has a spring

generated acceleration, a. We only apply the

drag if the is an acceleration (or force).

No drag is imposed if the spring is at rest.

\*/

static void applyDrag (unsigned int id, coord\_Coord a);

/\*

doApplySpringForce -

\*/

static coord\_Coord doApplySpringForce (unsigned int id, coord\_Coord force);

/\*

doApplyForce -

\*/

static void doApplyForce (unsigned int i, Object iptr);

/\*

applyForce - translate the force into acceleration

and update stationary boolean.

\*/

static void applyForce (void);

/\*

calcSpringEnergy -

\*/

static void calcSpringEnergy (unsigned int i);

/\*

calcObjectEnergy -

\*/

static void calcObjectEnergy (unsigned int i);

/\*

calcEnergy -

\*/

static void calcEnergy (void);

/\*

recalculateForceEnergy - recalculate all forces and energy

for all objects.

\*/

static void recalculateForceEnergy (void);

/\*

calcForce - calculate all forces for objects attached by springs.

\*/

static void calcForce (void);

/\*

calculateCofG -

\*/

static coord\_Coord calculateCofG (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high);

/\*

calcArea -

\*/

static double calcArea (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high);

/\*

debugCircle - displays a circle at position, p, with radius, r, in colour, c.

\*/

static void debugCircle (coord\_Coord p, double r, deviceIf\_Colour c);

/\*

debugLine - displays a line from, p1, to, p2, in the debugging colour.

\*/

static void debugLine (coord\_Coord p1, coord\_Coord p2, deviceIf\_Colour c);

/\*

doCircle - pass parameters to the groffDevice.

\*/

static void doCircle (coord\_Coord p, double r, deviceIf\_Colour c);

/\*

doPolygon -

\*/

static void doPolygon (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high, deviceIf\_Colour c);

/\*

drawBoarder -

\*/

static void drawBoarder (deviceIf\_Colour c);

/\*

drawBackground -

\*/

static void drawBackground (deviceIf\_Colour c);

/\*

getVelCoord - returns a velocity coordinate pair for Object, o.

\*/

static coord\_Coord getVelCoord (Object o);

/\*

getAccelCoord - returns an acceleration coordinate pair for Object, o.

\*/

static coord\_Coord getAccelCoord (Object o);

/\*

doCircleFrame -

\*/

static void doCircleFrame (Object optr, double dt, deviceIf\_Colour col);

/\*

doPolygonFrame -

\*/

static void doPolygonFrame (Object optr, double dt, deviceIf\_Colour col);

/\*

doSpringFrame -

\*/

static void doSpringFrame (Object optr, double dt, unsigned int col);

/\*

doDrawFrame -

\*/

static void doDrawFrame (Object optr, double dt, deviceIf\_Colour col);

/\*

getEventObjects -

\*/

static void getEventObjects (Object \*id1, Object \*id2, eventQueue e);

/\*

getColour -

\*/

static deviceIf\_Colour getColour (Object optr);

/\*

getSpringColour -

\*/

static deviceIf\_Colour getSpringColour (void);

/\*

getCollisionColour -

\*/

static deviceIf\_Colour getCollisionColour (void);

/\*

getEventObjectColour -

\*/

static deviceIf\_Colour getEventObjectColour (eventQueue e, Object optr);

/\*

equalBroadphase - return TRUE if the pair of objects in, b, is the same

as (i, j).

\*/

static unsigned int equalBroadphase (broadphase b, unsigned int i, unsigned int j);

/\*

assignBroadphase - assign each field in broadphase to: i, j and next.

b is returned.

\*/

static broadphase assignBroadphase (broadphase b, unsigned int i, unsigned int j, broadphase next);

/\*

newBroadphase - return a new initialised broadphase element.

\*/

static broadphase newBroadphase (unsigned int i, unsigned int j, broadphase next);

/\*

addBroadphase - adds, pair, i, j, onto the head of the broadphase list

as long as the pair is unique. It returns the new element

which is chained to head.

\*/

static broadphase addBroadphase (broadphase head, unsigned int i, unsigned int j);

/\*

initBroadphase - the constructor which returns a new broadphase list of objects.

\*/

static broadphase initBroadphase (void);

/\*

killBroadphase - returns list, head, back to the freeBroadphase list.

\*/

static void killBroadphase (broadphase \*head);

/\*

checkFrameBasedInterpen -

\*/

static void checkFrameBasedInterpen (void);

/\*

drawFrame - draws the current world into the frame buffer.

If e is not NIL then it will be a collision event

which describes the objects colliding. The

drawFrame will draw these objects using

the debugging colour.

\*/

static void drawFrame (eventQueue e);

/\*

drawFrameEvent -

\*/

static void drawFrameEvent (eventQueue e);

/\*

incRadians - return (a + b) mod 2pi. The value returned will be between 0..2pi

\*/

static double incRadians (double a, double b);

/\*

updatePolygon -

\*/

static void updatePolygon (Object optr, double dt);

/\*

updateCircle -

\*/

static void updateCircle (Object optr, double dt);

/\*

updateSpring - update the current length, l1, field of the spring.

\*/

static void updateSpring (Object optr, double dt);

/\*

updateOb -

\*/

static void updateOb (Object optr, double dt);

/\*

doUpdatePhysics - updates all positions of objects based on the passing of

dt seconds.

\*/

static void doUpdatePhysics (double dt);

/\*

updatePhysics - updates the velocity of all objects based on

the elapsed time from the last collision until now.

\*/

static void updatePhysics (unsigned int recalculateForce);

/\*

displayEvent -

\*/

static void displayEvent (eventQueue e);

/\*

printQueue - prints out the event queue.

\*/

static void printQueue (void);

/\*

updateStats -

\*/

static void updateStats (double dt);

/\*

doFunctionEvent -

\*/

static void doFunctionEvent (eventQueue e);

/\*

doNextEvent -

\*/

static double doNextEvent (void);

/\*

checkZero -

\*/

static void checkZero (double \*v);

/\*

checkZeroCoord -

\*/

static coord\_Coord checkZeroCoord (coord\_Coord c);

/\*

inElasticSpring -

\*/

static void inElasticSpring (double \*v);

/\*

inElastic -

\*/

static double inElastic (Object o, double v);

/\*

nearZeroVelocity - returns TRUE if, r, is close to 0.0

\*/

static unsigned int nearZeroVelocity (double r);

/\*

checkStationary - checks to see if object, o, should be put into

the stationary state.

\*/

static void checkStationary (Object o);

/\*

checkStationarySpring - checks to see if object, o, should be put into

the stationary state.

\*/

static void checkStationarySpring (Object o);

/\*

checkStationaryCollision - stationary object, a, has been bumped by

moving object, b. We move a slightly and

give it an initial velocity and change its

state from stationary to moving.

\*/

static void checkStationaryCollision (Object a, Object b);

/\*

collideFixedCircles - works out the new velocity given that the circle

movable collides with the fixed circle.

\*/

static void collideFixedCircles (Object movable, Object fixed);

/\*

collideAgainstFixedCircle - the movable object collides against a point, center.

center, is the center point of the other fixed circle.

This procedure works out the new velocity of the movable

circle given these constraints.

\*/

static void collideAgainstFixedCircle (Object movable, coord\_Coord center);

/\*

collideMovableCircles -

\*/

static void collideMovableCircles (Object iptr, Object jptr);

/\*

circleCollision - call fixed or movable circle collision depending upon whether

one or two circles are fixed.

Apart from taking into account rotation of either circle this

is complete.

\*/

static void circleCollision (Object iptr, Object jptr);

/\*

collideCircleAgainstFixedEdge - modifies the circle velocity based upon the edge it hits.

We use the formula:

V = 2 \* (-I . N ) \* N + I

where:

I is the initial velocity vector

V is the final velocity vector

N is the normal to the line

\*/

static void collideCircleAgainstFixedEdge (Object cPtr, coord\_Coord p1, coord\_Coord p2);

/\*

circlePolygonCollision -

\*/

static void circlePolygonCollision (eventQueue e, Object cPtr, Object pPtr);

/\*

collidePolygonAgainstFixedCircle - polygon, o, is moving and has hit

a fixed circle at position, collision.

\*/

static void collidePolygonAgainstFixedCircle (Object o, coord\_Coord collision);

/\*

collidePolygonAgainstFixedEdge - p1, p2 is the fixed edge and, o, is the

moving polygon.

\*/

static void collidePolygonAgainstFixedEdge (Object o, coord\_Coord p1, coord\_Coord p2);

/\*

rotationalVelocity - return a vector containing the velocity

if polygon is rotating at angular speed, w,

and non rotation speed is, u and the relative

position is, r.

\*/

static coord\_Coord rotationalVelocity (double w, coord\_Coord u, coord\_Coord r);

/\*

calcInertia - calculate the inertia of a polygon.

\*/

static void calcInertia (Object id);

/\*

updatePolygonVelocity - update both linear and angular velocities of

object, o, using impulse, j.

n is the perpendicular to the impact.

rp is the distance of the c of g of an object,

o, to the point of collision.

\*/

static void updatePolygonVelocity (Object o, double j, coord\_Coord n, coord\_Coord rpn);

/\*

polygonPolygonCollision - two polygons collide, we call the appropriate routines

depending upon whether one polygon is fixed.

\*/

static void polygonPolygonCollision (eventQueue e, Object id1, Object id2);

/\*

reflect - reflect velocity, v, off line, l.

V = 2 \* (-I . n ) \* n + I

where:

I is the initial velocity vector

V is the final velocity vector

n is the normal to the line, l.

\*/

static coord\_Coord reflect (coord\_Coord v, coord\_Coord l);

/\*

collidePolygonAgainstFixedPolygon - moving, and, fixed, are two polygons.

Work out the new velocity of the moving polygon

and also rotation velocity.

\*/

static void collidePolygonAgainstFixedPolygon (eventQueue e, Object moving, Object fixed);

/\*

collidePolygonAgainstMovingPolygon - both, id1, and, id2, are moving.

\*/

static void collidePolygonAgainstMovingPolygon (eventQueue e, Object id1, Object id2);

/\*

physicsCollision - handle the physics of a collision between

the two objects defined in, e.

\*/

static void physicsCollision (eventQueue e);

/\*

doCollision - called whenever a collision event is processed.

\*/

static void doCollision (eventQueue e);

/\*

sqr -

\*/

static double sqr (double v);

/\*

cub -

\*/

static double cub (double v);

/\*

quad -

\*/

static double quad (double v);

/\*

pent -

\*/

static double pent (double v);

/\*

hex -

\*/

static double hex (double v);

/\*

sept -

\*/

static double sept (double v);

/\*

oct -

\*/

static double oct (double v);

/\*

getCircleValues - assumes, o, is a circle and retrieves:

center (x, y)

radius radius

velocity (vx, vy)

accel (ax, ay)

\*/

static void getCircleValues (Object o, double \*x, double \*y, double \*radius, double \*vx, double \*vy, double \*ax, double \*ay);

/\*

getObjectValues - fills in velocity and acceleration x, y, values.

\*/

static void getObjectValues (Object o, double \*vx, double \*vy, double \*ax, double \*ay);

/\*

getObjectOrbitingValues -

\*/

static void getObjectOrbitingValues (Object o, double \*r, double \*w, coord\_Coord \*cofg);

/\*

maximaCircleCollisionOrbiting -

x1 y1 x2 y2

a, g, l, r is initial position of the point (not the c of g)

b, h, m, s is initial velocity

c, i, n, u is acceleration

e, k, p, v is angular velocity

f, q, k, w is the initial angular offset for the center of circle relative to the c of g.

The c of g is the center of the orbit.

d, o the distance of the point from the c of g.

\*/

static void maximaCircleCollisionOrbiting (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double i, double j, double k, double l, double m, double n, double o, double p, double q, double r, double s, double u, double v, double w, double x);

/\*

earlierCircleCollisionOrbiting -

t is the time of this collision (if any)

tc is the time of the next collision.

c1p is the initial position of the center of circle 1.

This may not be the c of g of this circle if it is in orbit.

c1radius is the radius of the circle.

c1r rotational offset of point 1.

c1cofg center of gravity of point 1.

c1w rotational angular velocity of point 1.

c1a acceleration of point 1.

c2p is the initial position of the center of circle 2.

This may not be the c of g of this circle if it is in orbit.

c2radius is the radius of the circle.

c2r rotational offset of point 2.

c2cofg center of gravity of point 2.

c2w rotational angular velocity of point 2.

c2a acceleration of point 2.

\*/

static unsigned int earlierCircleCollisionOrbiting (double \*t, double \*tc, coord\_Coord c1p, double c1radius, double c1r, double c1w, coord\_Coord c1cofg, coord\_Coord c1v, coord\_Coord c1a, coord\_Coord c2p, double c2radius, double c2r, double c2w, coord\_Coord c2cofg, coord\_Coord c2v, coord\_Coord c2a);

/\*

maximaCircleCollision -

\*/

static void maximaCircleCollision (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p);

static unsigned int earlierCircleCollision (eventDesc edesc, unsigned int id1, unsigned int id2, double \*t, double bestTimeOfCollision, coord\_Coord \*cp, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p);

/\*

doEarlierCircleCollision - let the following abreviations be assigned.

Note i is one circle, j is another circle.

v is velocity, a, acceleration, x, y axis.

r is radius.

Single letter variables are used since wxmaxima

only operates with these. Thus the output from wxmaxima

can be cut and pasted into the program.

a = xi

b = xj

c = vxi

d = vxj

e = aix

f = ajx

g = yi

h = yj

k = vyi

l = vyj

m = aiy

n = ajy

o = ri

p = rj

t is the time of this collision (if any)

bestTimeOfCollision is earlier known collision so far.

\*/

static unsigned int doEarlierCircleCollision (eventDesc edesc, unsigned int id1, unsigned int id2, double \*t, double bestTimeOfCollision, coord\_Coord \*cp, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p);

/\*

findCollisionCircles -

using:

S = UT + (AT^2)/2

compute xin and yin which are the new (x,y) positions of object i at time, t.

compute xjn and yjn which are the new (x,y) positions of object j at time, t.

now compute difference between objects and if they are ri+rj (radius of circle, i, and, j)

apart then we have a collision at time, t.

xin = xi + vxi \* t + (aix \* t^2) / 2.0

yin = yi + vyi \* t + (aiy \* t^2) / 2.0

xjn = xj + vxj \* t + (ajx \* t^2) / 2.0

yjn = yj + vyj \* t + (ajy \* t^2) / 2.0

ri + rj == sqrt(abs(xin-xjn)^2 + abs(yin-yjn)^2) for values of t

ri + rj == sqrt(((xi + vxi \* t + aix \* t^2 / 2.0) - (xj + vxj \* t + ajx \* t^2 / 2.0))^2 +

((yi + vyi \* t + aiy \* t^2 / 2.0) - (yj + vyj \* t + ajy \* t^2 / 2.0))^2)

let:

a = xi

b = xj

c = vxi

d = vxj

e = aix

f = ajx

g = yi

h = yj

k = vyi

l = vyj

m = aiy

n = ajy

o = ri

p = rj

t = t

o + p == sqrt(((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2)

o + p == sqrt(((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2)

0 == ((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2 -

(o + p)^2

now using wxmaxima

expand ; factor ; ratsimp

p+o == (sqrt((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))/2

2\*(p+o) == (sqrt((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))

(2\*(p+o))^2 == ((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))

0 == (n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+

4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2)-

((2\*(p+o))^2)

solve polynomial:

A := sqr(n)-2.0\*m\*n+sqr(m)+sqr(f)-2.0\*e\*f+sqr(e) ;

B := (4.0\*l-4.0\*k)\*n+(4.0\*k-4.0\*l)\*m+(4.0\*d-4.0\*c)\*f+(4.0\*c-4.0\*d)\*e ;

C := (4.0\*h-4.0\*g)\*n+(4.0\*g-4.0\*h)\*m+4.0\*sqr(l)-8.0\*k\*l+4.0\*sqr(k)+(4.0\*b-4.0\*a)\*f+(4.0\*a-4.0\*b)\*e+4.0\*sqr(d)-8.0\*c\*d+4.0\*sqr(c) ;

D := (8.0\*h-8.0\*g)\*l+(8.0\*g-8.0\*h)\*k+(8.0\*b-8.0\*a)\*d+(8.0\*a-8.0\*b)\*c ;

E := 4.0\*sqr(h)-8.0\*g\*h+4.0\*sqr(g)+4.0\*sqr(b)-8.0\*a\*b+4.0\*sqr(a)-sqr(2.0\*(p+o)) ;

\*/

static void findCollisionCircles (Object iptr, Object jptr, eventDesc \*edesc, double \*tc);

/\*

stop -

\*/

static void stop (void);

/\*

makeCirclesPolygonDesc - returns an eventDesc describing the collision between a circle and a polygon.

notice wpid1 is not used, this is because it is called indirectly

and there are other make...Desc functions which have exactly the

same parameter data types.

\*/

static eventDesc makeCirclesPolygonDesc (eventDesc edesc, unsigned int cid, unsigned int pid, unsigned int lineNo, unsigned int pointNo, history\_whereHit wpid1, history\_whereHit wpid2, coord\_Coord collisionPoint);

/\*

checkIfPointHits - return TRUE if t0 is either the first hit found or

is sooner than, tc. It determines a hit by working out

the final position of partical:

x = s + ut + 1/2a t^2

if x>=0.0 and x <= length then it hits.

\*/

static unsigned int checkIfPointHits (double timeOfPrevCollision, double t, double length, double s, double u, double a);

/\*

newPositionScalar - calculates the new position of a scalar in the future.

\*/

static double newPositionScalar (double s, double u, double a, double t);

/\*

newPositionRotationSinScalar - works out the new Y position for a point whose:

current cofg Y position is: c

initial Y velocity is : u

Y acceleration is : a

angular velocity : w

orientation : o

polar coord position rel

to cofg is : p

\*/

static double newPositionRotationSinScalar (double c, double u, double a, double t, double w, double o, polar\_Polar p);

/\*

newPositionRotationCosScalar - works out the new X position for a point whose:

current cofg X position is: c

initial X velocity is : u

X acceleration is : a

angular velocity : w

orientation : o

polar coord position rel

to cofg is : p

\*/

static double newPositionRotationCosScalar (double c, double u, double a, double t, double w, double o, polar\_Polar p);

/\*

newPositionCoord - calculates the new position of point in the future.

\*/

static coord\_Coord newPositionCoord (coord\_Coord c, coord\_Coord u, coord\_Coord a, double t);

/\*

newPositionRotationCoord - calculates the new position of point, c+v, in the future.

Given angular velocity : w

orientation : o

time : t

initial vel : u

accel : a

c of g : c

polar coord of the point : p

\*/

static coord\_Coord newPositionRotationCoord (coord\_Coord c, coord\_Coord u, coord\_Coord a, double t, double w, double o, polar\_Polar p);

/\*

hLine - debugging procedure to display a line on a half scale axis.

\*/

static void hLine (coord\_Coord p1, coord\_Coord p2, deviceIf\_Colour c);

/\*

hPoint - debugging procedure to display a line on a half scale axis.

\*/

static void hPoint (coord\_Coord p, deviceIf\_Colour c);

/\*

hCircle - debugging procedure to display a circle on a half scale axis.

\*/

static void hCircle (coord\_Coord p, double r, deviceIf\_Colour c);

/\*

hVec - display a normalised vector on a half scale axis

\*/

static void hVec (coord\_Coord p, deviceIf\_Colour c);

/\*

hFlush - flip the debugging buffer.

\*/

static void hFlush (void);

/\*

checkPointCollision -

\*/

static unsigned int checkPointCollision (double \*timeOfPrevCollision, double t, double length, double cx, double rvx, double rax, coord\_Coord c, coord\_Coord cvel, coord\_Coord caccel, coord\_Coord \*collisionPoint, unsigned int id1, unsigned int id2);

/\*

earlierPointLineCollision - returns TRUE if we can find a collision between a point, c,

travelling at cvel with acceleration, caccel and a line

p1, p2, travelling at velocity, lvel, and acceleration laccel.

If a collision is found then the collisionPoint is also

calculated.

\*/

static unsigned int earlierPointLineCollision (double \*timeOfCollision, coord\_Coord c, coord\_Coord cvel, coord\_Coord caccel, coord\_Coord p1, coord\_Coord p2, coord\_Coord lvel, coord\_Coord laccel, coord\_Coord \*collisionPoint, unsigned int id1, unsigned int id2);

/\*

sortLine - orders points, p1 and, p2, according to their x value.

\*/

static void sortLine (coord\_Coord \*p1, coord\_Coord \*p2);

/\*

findEarlierCircleEdgeCollision - return TRUE if an earlier time, t, is found than tc for when circle

hits a line. The circle is defined by a, center, radius and has

acceleration, accelCircle, and velocity, velCircle.

The line is between p1 and p2 and has velocity, velLine, and

acceleration, accelLine.

\*/

static void findEarlierCircleEdgeCollision (double \*timeOfCollision, unsigned int cid, unsigned int pid, unsigned int lineP, unsigned int lineC, eventDesc \*edesc, coord\_Coord center, double radius, coord\_Coord velCircle, coord\_Coord accelCircle, coord\_Coord p1, coord\_Coord p2, coord\_Coord velLine, coord\_Coord accelLine, descP createDesc);

/\*

getPolygonCoord -

\*/

static coord\_Coord getPolygonCoord (Object pPtr, unsigned int pointno);

/\*

getPolygonLine - assigns, c1, c2, with the, line, coordinates of polygon, pPtr.

\*/

static void getPolygonLine (unsigned int line, Object pPtr, coord\_Coord \*c1, coord\_Coord \*c2);

/\*

findCollisionCircleLine - find the time (if any) between line number, lineP, in polygon, pPtr,

and the circle, cPtr. cPtr can also be a polygon in which case lineC

is the particular line under question. Line on line collision is broken

down into circle line calls which allows for code reuse.

Neither cPtr or pPtr are orbiting.

\*/

static void findCollisionCircleLine (Object cPtr, Object pPtr, unsigned int lineP, unsigned int lineC, coord\_Coord center, double radius, eventDesc \*edesc, double \*timeOfCollision, descP createDesc);

/\*

findCollisionCircleLineOrbiting - find the time (if any) between line number, lineP, in polygon, pPtr,

and the circle, cPtr. cPtr can also be a polygon in which case lineC

is the particular line under question. Line on line collision is broken

down into circle line calls which allows for code reuse.

Either cPtr or pPtr or both are orbiting.

\*/

static void findCollisionCircleLineOrbiting (Object cPtr, Object pPtr, unsigned int lineP, unsigned int lineC, coord\_Coord center, double radius, eventDesc \*edesc, double \*timeOfCollision, descP createDesc);

/\*

findCollisionCirclePolygon - find the smallest positive time (if any) between the polygon and circle.

If a collision if found then, tc, is assigned to the time and cid, pid

are set to the circle id and polygon id respectively.

\*/

static void findCollisionCirclePolygon (Object cPtr, Object pPtr, eventDesc \*edesc, double \*tc);

/\*

makePolygonPolygon - return a new eventDesc indicating that we have a polygon/polygon collision

event.

\*/

static eventDesc makePolygonPolygon (eventDesc edesc, unsigned int id1, unsigned int id2, unsigned int lineCorner2, unsigned int lineCorner1, history\_whereHit wpid1, history\_whereHit wpid2, coord\_Coord collisionPoint);

/\*

isOrbiting - return TRUE if object, o, is rotating.

\*/

static unsigned int isOrbiting (Object o);

/\*

findCollisionLineLine - find the smallest time in the future when two lines collide.

The event descriptor, edesc, will contain the description of the collision

and, tc, the time of collision in the future.

\*/

static void findCollisionLineLine (Object iPtr, Object jPtr, unsigned int jLine, unsigned int iLine, eventDesc \*edesc, double \*tc);

/\*

findCollisionPointLine - determines whether point, p, which is at one of the ends of iPtr:iLine will hit

line jPtr:jLine. If so then the time of collision is recorded in, tc, (provided

it is sooner than the current value of tc). It also updates the event descriptor,

edesc.

\*/

static void findCollisionPointLine (Object iPtr, Object jPtr, unsigned int jLine, unsigned int iLine, coord\_Coord p, eventDesc \*edesc, double \*tc);

/\*

findCollisionPointLine - determines whether point, p, which is at one of the ends of iPtr:iLine will hit

line jPtr:jLine. If so then the time of collision is recorded in, tc, (provided

it is sooner than the current value of tc). It also updates the event descriptor,

edesc.

\*/

static void findCollisionLineLineNonOrbiting (Object iPtr, Object jPtr, unsigned int iLine, unsigned int jLine, eventDesc \*edesc, double \*tc);

/\*

Abs - return the absolute value of, r.

\*/

static double Abs (double r);

/\*

findAllTimesOfCollisionRLineRPoint -

4 4 4

- ((16 j k - 16 d e ) t

3 3 3 3 3

+ (64 j k l - 32 %pi j k - 64 d e f + 32 %pi d e ) t

2 2 2 2 2 2 2

+ (96 j k l - 96 %pi j k l + (24 %pi - 32) j k + 32 i - 96 d e f

2 2 2 2

+ 96 %pi d e f + (32 - 24 %pi ) d e - 32 c) t

3 2 2 3

+ (64 j k l - 96 %pi j k l + (48 %pi - 64) j k l + (32 %pi - 8 %pi ) j k

3 2 2

+ 64 h - 64 d e f + 96 %pi d e f + (64 - 48 %pi ) d e f

3 4 3

+ (8 %pi - 32 %pi) d e - 64 b) t + 16 j l - 32 %pi j l

2 2 3 4 2

+ (24 %pi - 32) j l + (32 %pi - 8 %pi ) j l + (%pi - 8 %pi + 64) j + 64 g

4 3 2 2 3

- 16 d f + 32 %pi d f + (32 - 24 %pi ) d f + (8 %pi - 32 %pi) d f

4 2

+ (- %pi + 8 %pi - 64) d - 64 a)/64 = 0

\*/

static unsigned int findAllTimesOfCollisionRLineRPoint (double a, double b, double c, double d, double e, double f, double g, double h, double i, double j, double k, double l, double \*t, unsigned int \_t\_high);

/\*

findEarliestCollisionRLineRPoint - find the earliest time when rotating point, j, collides with rotating line, i.

\*/

static void findEarliestCollisionRLineRPoint (Object iPtr, Object jPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc, double si, double ui, double ai, double ri, double wi, double oi, double sj, double uj, double aj, double rj, double wj, double oj, coord\_Coord p1, coord\_Coord p2);

/\*

findCollisionLineRPoint -

\*/

static void findCollisionLineRPoint (Object iPtr, Object rPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc);

/\*

findCollisionLineRLine - find the time of collision between line, iPtr, and rotating line, rPtr.

\*/

static void findCollisionLineRLine (Object iPtr, Object rPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc);

/\*

dumpLine -

\*/

static void dumpLine (unsigned int id, unsigned int line, history\_whereHit at);

/\*

dumpWhere -

\*/

static void dumpWhere (history\_whereHit at);

/\*

dumpDesc -

\*/

static void dumpDesc (eventDesc e);

/\*

findCollisionPolygonPolygon - find the smallest positive time (if any) between the polygons, iPtr

and jPtr colliding.

If a collision if found then, tc, is assigned to the time and the

event descriptor is filled in.

\*/

static void findCollisionPolygonPolygon (Object iPtr, Object jPtr, eventDesc \*edesc, double \*tc);

/\*

findCollisionPolygonRPolygon - find the smallest positive time (if any) between the polygons, iPtr

and rPtr colliding.

rPtr is a rotating polygon and iPtr is not rotating.

If a collision if found then, tc, is assigned to the time and the

event descriptor is filled in.

We check possible collision times between all lines of both polygons,

we separate out the rotating polygon from non rotating polygon

as the collision equations only generate a polynomial order 4 rather

than order 8 if both are rotating.

\*/

static void findCollisionPolygonRPolygon (Object iPtr, Object rPtr, eventDesc \*edesc, double \*tc);

/\*

findCollision -

\*/

static void findCollision (Object iptr, Object jptr, eventDesc \*edesc, double \*tc);

/\*

debugFrame - debug frame at time, e.

\*/

static void debugFrame (eventQueue e);

/\*

addDebugging - add a debugging event at time, t.

\*/

static void addDebugging (double t, eventDesc edesc);

/\*

anticipateCollision - stores the collision in the anticipated list.

\*/

static void anticipateCollision (double tc, eventDesc edesc);

/\*

collisionOccurred - stores the collision in the history list.

\*/

static void collisionOccurred (eventDesc edesc);

/\*

subEvent - remove event, e, from the relative time ordered event queue.

\*/

static void subEvent (eventQueue e);

/\*

removeCollisionEvent -

\*/

static void removeCollisionEvent (void);

/\*

removeSpringEvents - removes all spring events.

\*/

static void removeSpringEvents (void);

/\*

getSpringEndValues - it retrieves the:

CofG : c

velocity : v

acceleration: a

of object, o.

\*/

static void getSpringEndValues (unsigned int o, coord\_Coord \*c, coord\_Coord \*v, coord\_Coord \*a);

/\*

manualCircleCollision -

\*/

static void manualCircleCollision (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p);

/\*

earlierSpringLength - i is a spring.

c1, c2 are the c of g of the objects 1 and 2.

v1, v2 are the velocities of objects 1 and 2.

a1, a2 are the acceleration of objects 1 and 2.

Single letter variables are used since wxmaxima

only operates with these. Thus the output from wxmaxima

can be cut and pasted into the program.

a = c1.x

b = c1.y

c = v1.x

d = v1.y

e = a1.x

f = a1.y

g = c2.x

h = c2.y

k = v2.x

l = v2.y

m = a2.x

n = a2.y

t is the time of this collision (if any)

bestTimeOfCollision is earlier known collision so far.

\*/

static unsigned int earlierSpringLength (eventDesc edesc, unsigned int id, double \*t, double bestTime, coord\_Coord c1, coord\_Coord v1, coord\_Coord a1, coord\_Coord c2, coord\_Coord v2, coord\_Coord a2, double l, history\_springPoint sp);

/\*

makeSpringDesc - creates and fills in the spring descriptor.

\*/

static eventDesc makeSpringDesc (eventDesc edesc, unsigned int i, history\_springPoint stype);

/\*

calcSpringLengthEvents -

\*/

static void calcSpringLengthEvents (unsigned int i);

/\*

manualSpringVelocityZero -

\*/

static void manualSpringVelocityZero (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h);

/\*

earlierSpringEnd - records the earliest time in the future when the

relative velocity between the two bodies is zero.

\*/

static void earlierSpringEnd (unsigned int id, coord\_Coord v1, coord\_Coord a1, coord\_Coord v2, coord\_Coord a2);

/\*

calcSpringEndEvents - the spring reaches maximum compression or extension when the

relative velocity between the objects attached to the spring

is zero.

\*/

static void calcSpringEndEvents (unsigned int i);

/\*

calcSpringEndEventsKE - calcalate the time at which there is no Kinetic energy

in spring, i. This will be when the spring reaches

its end point.

\*/

static void calcSpringEndEventsKE (double \*ts, unsigned int i, eventDesc \*edesc);

/\*

calcSpringEventTime - calculates the time in the future when spring, i,

reaches its:

mid point and minimum or maximum extension.

Both events are stored as they may be independent.

If they are not independent, it wont matter as the

event queue will be recreated.

\*/

static void calcSpringEventTime (unsigned int i);

/\*

addSpringEvent -

\*/

static void addSpringEvent (double t, eventProc dop, eventDesc edesc);

/\*

reverseSpringAccel -

\*/

static void reverseSpringAccel (Object o);

/\*

zeroSpringAccel - assign (0.0, 0.0) to the acceration and force vectors.

\*/

static void zeroSpringAccel (Object o);

/\*

doSpringMidPoint - reached the mid point of the spring, reverse the

acceleration of the sprung objects.

\*/

static void doSpringMidPoint (eventQueue e);

/\*

doSpringEndPoint - reached the end point of the spring, we

remove some energy from the sprung objects.

\*/

static void doSpringEndPoint (eventQueue e);

/\*

doSpringCallPoint - reached the user defined call point.

We need to activate the call back.

\*/

static void doSpringCallPoint (eventQueue e);

/\*

doSpring - called whenever a spring event is processed.

\*/

static void doSpring (eventQueue e);

/\*

springOccurred - stores the spring event in the history list.

\*/

static void springOccurred (eventDesc edesc);

/\*

anticipateSpring - stores the collision in the anticipated list.

\*/

static void anticipateSpring (double tc, eventDesc edesc);

/\*

addNextSpringEvent -

\*/

static void addNextSpringEvent (void);

/\*

optPredictiveBroadphase - this function returns the list after culling

any pair objects which are moving in opposite

directions. It should check for velocity and

acceleration - making sure that both have the

same sign.

\*/

static broadphase optPredictiveBroadphase (broadphase list);

/\*

addNextCollisionEvent - recalculate the next collision event time.

\*/

static void addNextCollisionEvent (void);

/\*

determineFrameBased - return TRUE if we need to use frame based collision detection.

\*/

static unsigned int determineFrameBased (void);

/\*

doAddNextObjectEvent - removes the next spring and collision event and recalculates

the time of both events.

\*/

static void doAddNextObjectEvent (void);

/\*

addNextObjectEvent - check to see if we are using predictive event mode

and if so then add the next predictive collision event.

\*/

static void addNextObjectEvent (void);

/\*

skipFor - skip displaying any frames for, t, simulated seconds.

\*/

static void skipFor (double t);

/\*

resetQueue -

\*/

static void resetQueue (void);

/\*

disposeEvent - returns the event to the free queue.

\*/

static void disposeEvent (eventQueue e);

/\*

disposeDesc - returns the event desc to the free queue.

\*/

static void disposeDesc (eventDesc \*d);

/\*

newDesc - returns a new eventDesc.

\*/

static eventDesc newDesc (void);

/\*

newEvent - returns a new eventQueue.

\*/

static eventQueue newEvent (void);

/\*

makeCirclesDesc - return a eventDesc which describes two circles colliding.

\*/

static eventDesc makeCirclesDesc (eventDesc \*edesc, unsigned int cid1, unsigned int cid2, coord\_Coord cp);

/\*

addRelative - adds event, e, into the relative event queue.

\*/

static void addRelative (eventQueue e);

/\*

addEvent - adds an event which has no collision associated with it.

Typically this is a debugging event or display frame event.

\*/

static void addEvent (eventKind k, double t, eventProc dop);

/\*

assertMovement -

\*/

static void assertMovement (unsigned int id1, unsigned int id2, char \*message\_, unsigned int \_message\_high);

/\*

assertCollisionEvent - assert that the collision event consists of two non fixed objects.

\*/

static void assertCollisionEvent (eventDesc edesc);

/\*

addCollisionEvent - adds a collision event, the edesc is attached to the,

eventQueue, which is placed onto the eventQ.

\*/

static void addCollisionEvent (double t, eventProc dop, eventDesc edesc);

/\*

isEvent - return TRUE if the next event is of kind, k.

\*/

static unsigned int isEvent (eventKind k);

/\*

dumpTime -

\*/

static void dumpTime (void);

/\*

recordEvent -

\*/

static void recordEvent (void);

/\*

pumpQueue - prime the event queue with initial frame and collision events.

\*/

static void pumpQueue (void);

/\*

up - recreate the event queue.

The pair up/down must be used to shutdown

the event queue if the world is to be altered.

\*/

static void up (void);

/\*

down - shutdown the event queue.

\*/

static void down (void);

/\*

killQueue - destroys the event queue and returns events to the free list.

\*/

static void killQueue (void);

/\*

writeCircles -

\*/

static void writeCircles (cDesc c);

/\*

writeKind -

\*/

static void writeKind (history\_whereHit k);

/\*

writeCirclePolygon -

\*/

static void writeCirclePolygon (cpDesc c);

/\*

writePolygonPolygon -

\*/

static void writePolygonPolygon (ppDesc p);

/\*

writeFunction -

\*/

static void writeFunction (fcDesc fc);

/\*

writeSpring -

\*/

static void writeSpring (spDesc sp);

/\*

writeDesc -

\*/

static void writeDesc (eventDesc p);

/\*

writeEvent - writes the first event to the file.

\*/

static void writeEvent (eventQueue e);

/\*

memDump -

\*/

static void memDump (void \* a, unsigned int len);

/\*

checkOpened - checks to see of the MemStream file has been created and if not then

it is opened.

\*/

static void checkOpened (void);

/\*

Init -

\*/

static void Init (void);

/\*

Assert -

\*/

static void Assert (unsigned int b, unsigned int line)

{

if (! b)

{

libc\_printf ((char \*) "twoDsim.mod:%d:error assert failed\\n", 36, line);

}

/\*

exit (1);

HALT

\*/

}

/\*

AssertR -

\*/

static void AssertR (double a, double b)

{

if (! (roots\_nearZero (a-b)))

{

libc\_printf ((char \*) "error assert failed: %g should equal %g difference is %g\\n", 58, a, b, a-b);

}

}

/\*

AssertRFail -

\*/

static void AssertRFail (double a, double b)

{

if (! (roots\_nearZero (a-b)))

{

libc\_printf ((char \*) "error assert failed: %g should equal %g difference is %g\\n", 58, a, b, a-b);

libc\_exit (1);

}

}

/\*

AssertRDebug -

\*/

static void AssertRDebug (double a, double b, char \*message\_, unsigned int \_message\_high)

{

typedef struct \_T6\_a \_T6;

struct \_T6\_a { char array[10+1]; };

\_T6 copy;

char message[\_message\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (message, message\_, \_message\_high+1);

if (! (roots\_nearZero (a-b)))

{

StrLib\_StrCopy ((char \*) message, \_message\_high, (char \*) &copy.array[0], 10);

libc\_printf ((char \*) "%s failed %g should equal %g difference is %g\\n", 48, &copy, a, b, a-b);

}

}

/\*

dumpSpring -

\*/

static void dumpSpring (Object o)

{

libc\_printf ((char \*) "spring exists between object %d and object %d (at rest %g, Hook %g, current length %g", 85, o->s.id1, o->s.id2, o->s.l0, o->s.k, o->s.l1);

if (o->s.hasCallBackLength)

{

libc\_printf ((char \*) ", call back %g)\\n", 17, o->s.cbl);

}

else

{

libc\_printf ((char \*) ")\\n", 3);

}

libc\_printf ((char \*) " spring is responsible for force (%g, %g) and acceleration (%g, %g) on object %d\\n", 83, o->s.f1.x, o->s.f1.y, o->s.a1.x, o->s.a1.y, o->s.id1);

libc\_printf ((char \*) " spring is responsible for force (%g, %g) and acceleration (%g, %g) on object %d\\n", 83, o->s.f2.x, o->s.f2.y, o->s.a2.x, o->s.a2.y, o->s.id2);

}

/\*

dumpCircle -

\*/

static void dumpCircle (Object o)

{

libc\_printf ((char \*) "circle at (%g, %g) radius %g mass %g colour %d\\n", 48, o->c.pos.x, o->c.pos.y, o->c.r, o->c.mass, o->c.col);

}

/\*

dumpPolygon -

\*/

static void dumpPolygon (Object o)

{

unsigned int i;

coord\_Coord p0;

coord\_Coord p1;

coord\_Coord c0;

libc\_printf ((char \*) "polygon mass %g colour %d\\n", 27, o->p.mass, o->p.col);

libc\_printf ((char \*) " c of g (%g,%g)\\n", 19, o->p.cOfG.x, o->p.cOfG.y);

for (i=0; i<=o->p.nPoints-1; i++)

{

c0 = coord\_addCoord (o->p.cOfG, polar\_polarToCoord (polar\_rotatePolar ((polar\_Polar) o->p.points.array[i], o->angleOrientation)));

libc\_printf ((char \*) " point at (%g,%g)\\n", 20, c0.x, c0.y);

}

for (i=1; i<=o->p.nPoints; i++)

{

libc\_printf ((char \*) " %d line ", 10, i);

getPolygonLine (i, o, &p0, &p1);

dumpCoord (p0);

libc\_printf ((char \*) " -> ", 4);

dumpCoord (p1);

libc\_printf ((char \*) "\\n", 2);

}

}

/\*

checkDeleted -

\*/

static void checkDeleted (Object o)

{

if (o->deleted)

{

libc\_printf ((char \*) "object %d has been deleted, should not be accessing it now\\n", 60, o->id);

}

}

/\*

dumpObject -

\*/

static void dumpObject (Object o)

{

libc\_printf ((char \*) "object %d ", 10, o->id);

if (o->deleted)

{

libc\_printf ((char \*) "is deleted\\n", 12);

return;

}

else if (o->fixed)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "is fixed ", 9);

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "is movable ", 11);

if (o->stationary)

{

libc\_printf ((char \*) "but is now stationary ", 22);

}

else if (! (roots\_nearZero (o->angularVelocity)))

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "and has a rotating velocity of %g\\n", 35, o->angularVelocity);

}

else if (! (roots\_nearZero (o->angleOrientation)))

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "and its current orientation is %g\\n", 35, o->angleOrientation);

}

}

switch (o->object)

{

case circleOb:

dumpCircle (o);

break;

case polygonOb:

dumpPolygon (o);

break;

case springOb:

dumpSpring (o);

break;

default:

break;

}

if (! o->fixed && ! o->stationary)

{

libc\_printf ((char \*) " velocity (%g, %g) acceleration (%g, %g)", 43, o->vx, o->vy, o->ax, o->ay);

libc\_printf ((char \*) " forces (%g, %g) spring acceleration (%g, %g) object gravity (%g)\\n", 67, o->forceVec.x, o->forceVec.y, o->saccel.x, o->saccel.y, o->gravity);

}

}

/\*

safeCoord - ensures that 0.0 <= r <= 1.0.

\*/

static double safeCoord (double r)

{

if (r < 0.0)

{

return 0.0;

}

else if (r > 1.0)

{

/\* avoid dangling else. \*/

return 1.0;

}

else

{

/\* avoid dangling else. \*/

return r;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

c2p - returns a Point given a Coord.

\*/

static Points\_Point c2p (coord\_Coord c)

{

return Points\_initPoint (Fractions\_putReal (safeCoord (c.x)), Fractions\_putReal (safeCoord (c.y)));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newObject - creates an object of, type, and returns its, id.

\*/

static unsigned int newObject (ObjectType type)

{

Object optr;

maxId += 1;

Storage\_ALLOCATE ((void \*\*) &optr, sizeof (\_T2));

optr->id = maxId;

optr->deleted = FALSE;

optr->fixed = FALSE;

optr->stationary = FALSE;

optr->visible = TRUE;

optr->saccel = coord\_initCoord (0.0, 0.0);

optr->gravity = 0.0;

optr->elasticity = Elasticity;

optr->forceVec = coord\_initCoord (0.0, 0.0);

optr->object = type;

optr->vx = 0.0;

optr->vy = 0.0;

optr->ax = 0.0;

optr->ay = 0.0;

optr->angularVelocity = 0.0;

optr->angularMomentum = 0.0;

optr->angleOrientation = 0.0;

optr->inertia = 0.0;

optr->interpen = 0;

Indexing\_PutIndice (objects, maxId, (void \*) optr);

return maxId;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

circle\_moving\_towards - return TRUE if object, optr, is moving towards

point, x, y. The object may still miss point x, y

but it is moving closer to this position.

\*/

static unsigned int circle\_moving\_towards (Object optr, double x, double y)

{

double contactVel;

coord\_Coord relativePosition;

coord\_Coord relativeVelocity;

relativePosition = coord\_initCoord (optr->c.pos.x-x, optr->c.pos.y-y);

relativeVelocity = coord\_initCoord (optr->vx, optr->vy);

contactVel = coord\_dotProd (relativeVelocity, relativePosition);

return contactVel < 0.0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

apply\_impulse\_to\_circle - apply an impulse to moving circle, movable,

along the vector [x, y] with magnitude, m.

\*/

static void apply\_impulse\_to\_circle (Object movable, double x, double y, double m)

{

double contactVel;

double theta;

double r;

double j;

coord\_Coord c;

coord\_Coord impulsePos;

coord\_Coord unitCollision;

coord\_Coord relativeVelocity;

/\* gdbif.sleepSpin ; \*/

impulsePos.x = movable->c.pos.x;

impulsePos.y = movable->c.pos.y;

if (roots\_nearZero (x))

{

if (y > 0.0)

{

impulsePos.y = impulsePos.y+movable->c.r;

}

else

{

impulsePos.y = impulsePos.y-movable->c.r;

}

}

else if (roots\_nearZero (y))

{

/\* avoid dangling else. \*/

if (x > 0.0)

{

impulsePos.x = impulsePos.x+movable->c.r;

}

else

{

impulsePos.x = impulsePos.x-movable->c.r;

}

}

else

{

/\* avoid dangling else. \*/

if ((x > 0.0) && (y > 0.0))

{

theta = libm\_atan (y/x);

impulsePos.x = impulsePos.x+((libm\_cos (theta))\*movable->c.r);

impulsePos.y = impulsePos.y+((libm\_sin (theta))\*movable->c.r);

}

else if ((x < 0.0) && (y < 0.0))

{

/\* avoid dangling else. \*/

x = -x;

y = -y;

theta = libm\_atan (y/x);

impulsePos.x = impulsePos.x-((libm\_cos (theta))\*movable->c.r);

impulsePos.y = impulsePos.y-((libm\_sin (theta))\*movable->c.r);

}

else if ((x > 0.0) && (y < 0.0))

{

/\* avoid dangling else. \*/

y = -y;

theta = libm\_atan (y/x);

impulsePos.x = impulsePos.x+((libm\_cos (theta))\*movable->c.r);

impulsePos.y = impulsePos.y-((libm\_sin (theta))\*movable->c.r);

}

else

{

/\* avoid dangling else. \*/

x = -x;

theta = libm\_atan (y/x);

impulsePos.x = impulsePos.x-((libm\_cos (theta))\*movable->c.r);

impulsePos.y = impulsePos.y+((libm\_sin (theta))\*movable->c.r);

}

}

c = coord\_initCoord (movable->c.pos.x-impulsePos.x, movable->c.pos.y-impulsePos.y);

/\*

frameNote ;

drawFrame (NIL) ;

debugCircle (impulsePos, 0.02, white ()) ;

\*/

r = libm\_sqrt ((c.x\*c.x)+(c.y\*c.y));

unitCollision = coord\_initCoord (c.x/r, c.y/r);

relativeVelocity = coord\_initCoord (movable->vx, movable->vy);

/\*

debugLine (impulsePos, addCoord (impulsePos, c), yellow ()) ;

flipBuffer ;

\*/

contactVel = coord\_dotProd (relativeVelocity, c);

if (contactVel < 0.0)

{

/\* moving towards. \*/

j = (-((1.0+1.0)\*((relativeVelocity.x\*unitCollision.x)+(relativeVelocity.y\*unitCollision.y))))/(((unitCollision.x\*unitCollision.x)+(unitCollision.y\*unitCollision.y))\*(1.0/movable->c.mass));

movable->vx = movable->vx+((j\*unitCollision.x)/movable->c.mass);

movable->vy = movable->vy+((j\*unitCollision.y)/movable->c.mass);

}

checkStationary (movable);

}

/\*

objectExists - returns TRUE if object, o, has not been deleted.

\*/

static unsigned int objectExists (Object o)

{

return (o != NULL) && ! o->deleted;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

objectIdExists - returns TRUE if object, id, has not been deleted.

\*/

static unsigned int objectIdExists (unsigned int id)

{

return objectExists ((Object) Indexing\_GetIndice (objects, id));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCheckInterpenCircleCircle -

\*/

static unsigned int doCheckInterpenCircleCircle (Object fixed, Object movable)

{

coord\_Coord d;

coord\_Coord v;

double h1;

double h0;

d = coord\_subCoord (movable->c.pos, fixed->c.pos);

h0 = fixed->c.r+movable->c.r;

h1 = libm\_sqrt ((d.x\*d.x)+(d.y\*d.y));

if (h0 > h1)

{

if (trace)

{

libc\_printf ((char \*) "interpen found two moving circles interpenetrating %d, %d h0 = %g, h1 = %g\\n", 78, fixed->id, movable->id, h0, h1);

}

/\* adjust movable circle. \*/

v = coord\_scaleCoord (coord\_normaliseCoord (d), h0);

movable->c.pos = coord\_addCoord (fixed->c.pos, v);

/\* checkStationary (movable) ; \*/

movable->vx = movable->vx+((v.x\*(h0-h1))/h0);

movable->vy = movable->vy+((v.y\*(h0-h1))/h0); /\* give it a little push as well. \*/

movable->stationary = FALSE; /\* give it a little push as well. \*/

movable->interpen += 1;

return movable->interpen;

}

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCheckInterpenCircleCircleMoving -

\*/

static unsigned int doCheckInterpenCircleCircleMoving (Object c1, Object c2)

{

coord\_Coord d;

coord\_Coord v;

double h1;

double h0;

d = coord\_subCoord (c2->c.pos, c1->c.pos);

h0 = c1->c.r+c2->c.r;

h1 = libm\_sqrt ((d.x\*d.x)+(d.y\*d.y));

if (h0 > h1)

{

if (trace)

{

libc\_printf ((char \*) "interpen found two moving circles interpenetrating %d, %d h0 = %g, h1 = %g\\n", 78, c1->id, c2->id, h0, h1);

}

/\* we should really adjust the circle with the lowest interpen value. \*/

v = coord\_scaleCoord (coord\_normaliseCoord (d), h0);

c2->c.pos = coord\_addCoord (c1->c.pos, v);

c2->vx = c2->vx+((v.x\*(h0-h1))/h0);

c2->vy = c2->vy+((v.y\*(h0-h1))/h0); /\* give it a little push as well. \*/

c2->stationary = FALSE; /\* give it a little push as well. \*/

c1->stationary = FALSE;

/\* checkStationary (c1) ; \*/

c2->interpen += 1;

return c2->interpen;

}

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

distanceLinePoint - c is a point. p1->p2 is the line in question.

p3 is assigned to the closest point on the line

to the point, c. d is the distance from c to p3.

TRUE is returned if the point, c, lies above or

below the line once the line is rotated onto the x

axis. (The point, c, would also rotated to solve

this question).

\*/

static unsigned int distanceLinePoint (coord\_Coord c, coord\_Coord p1, coord\_Coord p2, coord\_Coord \*p3, double \*d)

{

double A;

double B;

double C;

double D;

double dot;

double lengthSq;

double normalised;

A = c.x-p1.x;

B = c.y-p1.y;

C = p2.x-p1.x;

D = p2.y-p1.y;

dot = (A\*C)+(B\*D);

lengthSq = (sqr (C))+(sqr (D));

normalised = -1.0;

if (! (roots\_nearZero (lengthSq)))

{

/\* the dot product divided by length squared

gives you the projection distance from p1.

This is the fraction of the line that the point c

is the closest. \*/

normalised = dot/lengthSq;

}

if (normalised < 0.0)

{

/\* misses line. \*/

(\*p3) = p1;

return FALSE;

}

else if (normalised > 1.0)

{

/\* avoid dangling else. \*/

/\* misses line. \*/

(\*p3) = p2;

return FALSE;

}

(\*p3) = checkZeroCoord (coord\_initCoord (p1.x+(normalised\*C), p1.y+(normalised\*D)));

(\*d) = coord\_lengthCoord (coord\_subCoord (c, (\*p3)));

return TRUE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkLimits -

\*/

static coord\_Coord checkLimits (coord\_Coord c, double r)

{

if ((c.x-r) < 0.0)

{

c.x = r;

}

else if ((c.x+r) > 1.0)

{

/\* avoid dangling else. \*/

c.x = 1.0-r;

}

if ((c.y-r) < 0.0)

{

c.y = r;

}

else if ((c.y+r) > 1.0)

{

/\* avoid dangling else. \*/

c.y = 1.0-r;

}

return c;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCheckInterpenCirclePolygon -

\*/

static unsigned int doCheckInterpenCirclePolygon (Object iptr, Object jptr)

{

double d;

double r;

unsigned int i;

unsigned int n;

coord\_Coord v;

coord\_Coord c;

coord\_Coord p1;

coord\_Coord p2;

coord\_Coord p3;

Assert (iptr->object == circleOb, 1267);

Assert (jptr->object == polygonOb, 1268);

c = checkZeroCoord (iptr->c.pos);

r = iptr->c.r;

n = jptr->p.nPoints;

i = 1;

while (i <= n)

{

getPolygonLine (i, jptr, &p1, &p2);

if (((distanceLinePoint (c, p1, p2, &p3, &d)) && (! (roots\_nearZero (r-d)))) && (r > d))

{

/\* circle collides with line and point, p3, is the closest

point on line, p1->p2 to, c. \*/

if (! iptr->fixed)

{

/\* circle is not fixed, move it. \*/

if (roots\_nearZero (d))

{

/\* avoid dangling else. \*/

/\*

v := subCoord (jptr^.p.cOfG, p3) ;

d := lengthCoord (v) ;

v := scaleCoord (v, (r+d)/d) ;

iptr^.c.pos := addCoord (jptr^.p.cOfG, v) ;

\*/

if (trace)

{

libc\_printf ((char \*) "distance is nearzero, seen collision between circle and line, new position %g, %g\\n", 83, iptr->c.pos.x, iptr->c.pos.y);

}

}

else

{

/\*

printf ("line p1 = %g, %g -> %g, %g and point %g, %g nearest point %g, %g

",

p1.x, p1.y, p2.x, p2.y, c.x, c.y, p3.x, p3.y) ;

printf ("seen collision between circle and line, adjusting %d

",

iptr^.id) ;

printf ("radius = %g, distance = %g

", r, d) ;

printf ("seen collision between circle and line, old position %g, %g

",

iptr^.c.pos.x, iptr^.c.pos.y) ;

\*/

v = coord\_subCoord (c, p3);

v = checkZeroCoord (coord\_scaleCoord (v, r/d));

/\*

printf ("v = %g, %g p3 = %g, %g

", v.x, v.y, p3.x, p3.y) ;

\*/

iptr->c.pos = checkZeroCoord (coord\_addCoord (p3, v));

}

/\*

IF iptr^.stationary

THEN

printf ("seen collision between circle and line, new position %g, %g (now stationary)

",

iptr^.c.pos.x, iptr^.c.pos.y)

ELSE

iptr^.vx := iptr^.vx - v.x ;

iptr^.vy := iptr^.vy - v.y ; give it a little push as well.

printf ("seen collision between circle and line, new position %g, %g, velocity %g, %g (pushing it by: %g, %g)

",

iptr^.c.pos.x, iptr^.c.pos.y, iptr^.vx, iptr^.vy, v.x, v.y)

END

\*/

iptr->c.pos = checkLimits (iptr->c.pos, r);

iptr->interpen += 1;

return iptr->interpen;

}

}

i += 1;

}

iptr->c.pos = checkLimits (iptr->c.pos, r);

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCheckInterpenPolygon -

\*/

static unsigned int doCheckInterpenPolygon (Object iptr, Object jptr)

{

if ((iptr->object == circleOb) && (jptr->object == polygonOb))

{

return doCheckInterpenCirclePolygon (iptr, jptr);

}

else if ((iptr->object == polygonOb) && (jptr->object == circleOb))

{

/\* avoid dangling else. \*/

return doCheckInterpenCirclePolygon (jptr, iptr);

}

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCheckInterpenCircle -

\*/

static unsigned int doCheckInterpenCircle (Object iptr, Object jptr)

{

if ((iptr->object == circleOb) && (jptr->object == circleOb))

{

/\* avoid gcc warning by using compound statement even if not strictly necessary. \*/

if (iptr->fixed && ! jptr->fixed)

{

return doCheckInterpenCircleCircle (iptr, jptr);

}

else if (! iptr->fixed && jptr->fixed)

{

/\* avoid dangling else. \*/

return doCheckInterpenCircleCircle (jptr, iptr);

}

else

{

/\* avoid dangling else. \*/

return doCheckInterpenCircleCircleMoving (iptr, jptr);

}

}

return 0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

initInterpen -

\*/

static void initInterpen (void)

{

unsigned int n;

unsigned int i;

Object iptr;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

iptr->interpen = 0;

}

i += 1;

}

}

/\*

max - return the maximum of a and b.

\*/

static unsigned int max (unsigned int a, unsigned int b)

{

if (a > b)

{

return a;

}

else

{

return b;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkMicroInterpenCircle -

\*/

static void checkMicroInterpenCircle (void)

{

unsigned int n;

unsigned int i;

unsigned int j;

unsigned int c;

Object iptr;

Object jptr;

initInterpen ();

n = Indexing\_HighIndice (objects);

do {

c = 0;

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

j = i+1;

while (j <= n)

{

jptr = Indexing\_GetIndice (objects, j);

if (objectExists (jptr))

{

c = max (doCheckInterpenCircle (iptr, jptr), c);

}

j += 1;

}

}

i += 1;

}

/\* keep going until no interpentration was found or there is a cycle found. \*/

} while (! ((c >= n) || (c == 0)));

}

/\*

checkMicroInterpenPolygon -

\*/

static void checkMicroInterpenPolygon (void)

{

unsigned int n;

unsigned int i;

unsigned int j;

unsigned int c;

Object iptr;

Object jptr;

initInterpen ();

n = Indexing\_HighIndice (objects);

do {

c = 0;

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

j = i+1;

while (j <= n)

{

jptr = Indexing\_GetIndice (objects, j);

if (objectExists (jptr))

{

c = max (doCheckInterpenPolygon (iptr, jptr), c);

}

j += 1;

}

}

i += 1;

}

/\* keep going until no interpentration was found or there is a cycle found. \*/

} while (! ((c >= n) || (c == 0)));

}

/\*

checkMicroInterpen - this performs micro collision analysis, it detects interpenetration

between objects and separates the objects, without much force.

This is called if we are using collision prediction and this will solve

rounding errors which might otherwise allow objects to fall into each other.

It will keep moving objects apart from each other and keep going in cycles

(for a limited number of cycles).

\*/

static void checkMicroInterpen (void)

{

/\* firstly we move circles away from polygons. \*/

checkMicroInterpenPolygon ();

/\* then we move circles away from circles. \*/

checkMicroInterpenCircle ();

}

/\*

resetStationary -

\*/

static void resetStationary (void)

{

unsigned int n;

unsigned int i;

Object iptr;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

if (iptr->stationary)

{

/\* unset stationary, but ensure velocity is zero. \*/

iptr->vx = 0.0;

iptr->vy = 0.0;

iptr->stationary = FALSE;

}

}

i += 1;

}

}

/\*

getInterCircle - return the interCircle data structure filled in from circle0.

\*/

static interpen\_interCircle getInterCircle (Object circle0)

{

interpen\_interCircle ic;

ic.radius = circle0->c.r;

ic.center = circle0->c.pos;

return ic;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkFrameInterpenCircleCircle -

\*/

static void checkFrameInterpenCircleCircle (Object circle0, Object circle1)

{

interpen\_interCircle c0;

interpen\_interCircle c1;

coord\_Coord p;

eventDesc edesc;

Assert (circle0->object == circleOb, 1565);

Assert (circle1->object == circleOb, 1566);

c0 = getInterCircle (circle0);

c1 = getInterCircle (circle1);

if (interpen\_circleCollide (c0, c1))

{

if (circle0->fixed)

{

/\* move circle1 away from circle0. \*/

p = coord\_normaliseCoord (coord\_subCoord (c1.center, c0.center));

circle1->c.pos = coord\_addCoord (c0.center, coord\_scaleCoord (p, c0.radius+c1.radius));

p = coord\_addCoord (c0.center, coord\_scaleCoord (p, c0.radius));

}

else

{

/\* move circle0 away from circle1. \*/

p = coord\_normaliseCoord (coord\_subCoord (c0.center, c1.center));

circle1->c.pos = coord\_addCoord (c1.center, coord\_scaleCoord (p, c1.radius+c0.radius));

p = coord\_addCoord (c1.center, coord\_scaleCoord (p, c1.radius));

}

if (! (history\_isDuplicateC (currentTime, 0.0, circle0->id, circle1->id, (history\_whereHit) history\_edge, (history\_whereHit) history\_edge, p)))

{

edesc = NULL;

edesc = makeCirclesDesc (&edesc, circle0->id, circle1->id, p);

addCollisionEvent (0.0, (eventProc) {(eventProc\_t) doCollision}, edesc);

}

}

}

/\*

checkFrameInterpenCirclePolygon -

\*/

static void checkFrameInterpenCirclePolygon (Object circle0, Object polygon0)

{

segment\_Segment s0;

interpen\_interCircle c0;

coord\_Coord p0;

coord\_Coord p1;

coord\_Coord p;

eventDesc edesc;

history\_whereHit at;

unsigned int ptn;

unsigned int i;

unsigned int n;

Assert (circle0->object == circleOb, 1610);

Assert (polygon0->object == polygonOb, 1611);

c0 = getInterCircle (circle0);

n = polygon0->p.nPoints;

i = 1;

while (i <= n)

{

getPolygonLine (i, polygon0, &p0, &p1);

s0 = segment\_initSegment (p0, p1);

if (interpen\_circleSegmentCollide (c0, s0, &p, &at, &ptn))

{

/\* --fixme-- do we now need to move the objects apart? \*/

if (! (history\_isDuplicateC (currentTime, 0.0, circle0->id, polygon0->id, at, at, p)))

{

/\* add collision event. \*/

edesc = NULL;

edesc = makeCirclesPolygonDesc (edesc, circle0->id, polygon0->id, i, i+ptn, at, at, p);

addCollisionEvent (0.0, (eventProc) {(eventProc\_t) doCollision}, edesc);

return;

}

}

i += 1;

}

}

/\*

dumpCoord -

\*/

static void dumpCoord (coord\_Coord p)

{

libc\_printf ((char \*) "(%g,%g)", 7, p.x, p.y);

}

/\*

dumpCollision -

\*/

static void dumpCollision (unsigned int pid0, unsigned int pid1, unsigned int l0, unsigned int l1, history\_whereHit at0, history\_whereHit at1, coord\_Coord p)

{

libc\_printf ((char \*) "polygon %d:%d vs polygon %d:%d ", 31, pid0, l0, pid1, l1);

dumpWhere (at0);

libc\_printf ((char \*) " ", 1);

dumpWhere (at1);

libc\_printf ((char \*) " ", 1);

dumpCoord (p);

libc\_printf ((char \*) "\\n", 2);

}

/\*

debugDelay -

\*/

static void debugDelay (char \*message\_, unsigned int \_message\_high)

{

int r;

char message[\_message\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (message, message\_, \_message\_high+1);

libc\_printf ((char \*) "debug delay: %s\\n", 18, message);

r = libc\_system ("sleep 3");

}

/\*

restoreOldCofG -

\*/

static void restoreOldCofG (Object poly)

{

if (! poly->fixed)

{

poly->p.cOfG = poly->p.oldcOfG;

}

}

/\*

checkFrameInterpenPolygonPolygon - checks every line segment of polygon0 vs polygon1

and registers a collision event at the current time

if these segments intersect.

\*/

static void checkFrameInterpenPolygonPolygon (Object polygon0, Object polygon1)

{

coord\_Coord p0;

coord\_Coord p1;

coord\_Coord p;

eventDesc edesc;

history\_whereHit at0;

history\_whereHit at1;

segment\_Segment s0;

segment\_Segment s1;

unsigned int ptn0;

unsigned int ptn1;

unsigned int i0;

unsigned int i1;

unsigned int n0;

unsigned int n1;

int r;

/\* --fixme-- you might want to improve this code. mcomp. \*/

Assert (polygon0->object == polygonOb, 1709);

Assert (polygon1->object == polygonOb, 1710);

n0 = polygon0->p.nPoints;

n1 = polygon1->p.nPoints;

i0 = 1;

if (PolygonDebugging)

{

libc\_printf ((char \*) "checkFrameInterpenPolygonPolygon n0 = %d, n1 = %d\\n", 51, n0, n1);

}

while (i0 <= n0)

{

getPolygonLine (i0, polygon0, &p0, &p1);

s0 = segment\_initSegment (p0, p1);

i1 = 1;

while (i1 <= n1)

{

getPolygonLine (i1, polygon1, &p0, &p1);

s1 = segment\_initSegment (p0, p1);

if (PolygonDebugging)

{

libc\_printf ((char \*) "polygon %d:%d vs polygon %d:%d\\n", 32, polygon0->id, i0, polygon1->id, i1);

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

libc\_printf ((char \*) " yellow coordinate pairs: %g, %g -> %g, %g\\n", 45, s0.point1, s0.point2);

libc\_printf ((char \*) " white coordinate pairs: %g, %g -> %g, %g\\n", 44, s1.point1, s1.point2);

debugLine (s0.point1, s0.point2, (deviceIf\_Colour) deviceIf\_yellow ());

debugLine (s1.point1, s1.point2, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

if (interpen\_segmentsCollide (s0, s1, &p, &at0, &at1, &ptn0, &ptn1))

{

/\* --fixme-- do we now need to move the objects apart? \*/

if (! (history\_isDuplicateC (currentTime, 0.0, polygon0->id, polygon1->id, at0, at1, p)))

{

/\* add collision event. \*/

if (PolygonDebugging)

{

libc\_printf ((char \*) "frame number %d: ", 17, deviceIf\_getFrameNo ());

libc\_printf ((char \*) "short circuit further test\\n", 28);

dumpCollision (polygon0->id, polygon1->id, i0, i1, at0, at1, p);

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

libc\_printf ((char \*) " yellow coordinate pairs: %g, %g -> %g, %g\\n", 45, s0.point1, s0.point2);

libc\_printf ((char \*) " white coordinate pairs: %g, %g -> %g, %g\\n", 44, s1.point1, s1.point2);

debugLine (s0.point1, s0.point2, (deviceIf\_Colour) deviceIf\_purple ());

debugLine (s1.point1, s1.point2, (deviceIf\_Colour) deviceIf\_purple ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

edesc = NULL;

edesc = makePolygonPolygon (edesc, polygon0->id, polygon1->id, i0+ptn0, i1+ptn1, at0, at1, p);

addCollisionEvent (0.0, (eventProc) {(eventProc\_t) doCollision}, edesc);

if (PolygonDebugging)

{

libc\_printf ((char \*) "interpen created queue\\n", 24);

printQueue ();

}

}

/\*

RETURN

\*/

}

i1 += 1;

}

i0 += 1;

}

}

/\*

checkFrameInterpenObjects -

\*/

static void checkFrameInterpenObjects (unsigned int i, unsigned int j)

{

Object iptr;

Object jptr;

iptr = Indexing\_GetIndice (objects, i);

jptr = Indexing\_GetIndice (objects, j);

if ((iptr->object == circleOb) && (jptr->object == circleOb))

{

checkFrameInterpenCircleCircle (iptr, jptr);

}

else if ((iptr->object == circleOb) && (jptr->object == polygonOb))

{

/\* avoid dangling else. \*/

checkFrameInterpenCirclePolygon (iptr, jptr);

}

else if ((iptr->object == polygonOb) && (jptr->object == circleOb))

{

/\* avoid dangling else. \*/

checkFrameInterpenCirclePolygon (jptr, iptr);

}

else if ((iptr->object == polygonOb) && (jptr->object == polygonOb))

{

/\* avoid dangling else. \*/

checkFrameInterpenPolygonPolygon (iptr, jptr);

}

}

/\*

displayBroadphase -

\*/

static void displayBroadphase (broadphase b)

{

libc\_printf ((char \*) " objects: %d and %d\\n", 23, b->o0, b->o1);

}

/\*

dumpBroadphase -

\*/

static void dumpBroadphase (broadphase list)

{

broadphase b;

libc\_printf ((char \*) "broadphase list:\\n", 18);

if (list == NULL)

{

libc\_printf ((char \*) " empty", 7);

}

else

{

b = list;

while (b != NULL)

{

displayBroadphase (b);

b = b->next;

}

}

}

/\*

optBroadphase - optimise the broadphase. (--fixme--) complete me.

\*/

static broadphase optBroadphase (broadphase list)

{

/\* your code goes here. mcomp. \*/

return list;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkFrameInterpen - at this point the engine is running in frame based mode

as we have at least one polygon moving and rotating.

We need to check for interpenetration of objects, move them apart

and add a collision event for each pair of interpenetrating objects.

\*/

static void checkFrameInterpen (void)

{

broadphase list;

broadphase b;

list = optBroadphase (initBroadphase ());

if (BroadphaseDebugging)

{

dumpBroadphase (list);

}

b = list;

while (b != NULL)

{

checkFrameInterpenObjects (b->o0, b->o1);

b = b->next;

}

killBroadphase (&list);

if (BroadphaseDebugging)

{

printQueue ();

}

}

/\*

getCofG - returns the CofG of an object.

\*/

static coord\_Coord getCofG (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

switch (idp->object)

{

case circleOb:

return idp->c.pos;

break;

case polygonOb:

return idp->p.cOfG;

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isCircle - return TRUE if object, id, is a circle.

\*/

static unsigned int isCircle (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

return idp->object == circleOb;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isPolygon - return TRUE if object, id, is a polygon.

\*/

static unsigned int isPolygon (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

return idp->object == polygonOb;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isSpringObject - return TRUE if object, id, is a spring.

\*/

static unsigned int isSpringObject (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

return (idp != NULL) && (idp->object == springOb);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

calcSpringFixed - calculate the forces on, moving object which is attached to, fixed.

Given spring properties of, k, and, l0.

\*/

static void calcSpringFixed (double k, double d, double l0, double l1, unsigned int spr, unsigned int fixed, unsigned int moving)

{

if (l1 > l0)

{

doCalcSpringFixed (k, d, l0, l1, spr, fixed, moving);

}

}

/\*

doCalcSpringFixed - calculate the forces on, moving object which is attached to, fixed.

Given spring properties of, k, and, l0.

\*/

static void doCalcSpringFixed (double k, double d, double l0, double l1, unsigned int spr, unsigned int fixed, unsigned int moving)

{

Object sprp;

Object fixedp;

Object movingp;

Object o;

coord\_Coord f1;

coord\_Coord vfm;

coord\_Coord fvec;

coord\_Coord s;

coord\_Coord n;

double factor;

double springval;

double damping;

double fax;

double fay;

double fvx;

double fvy;

double mvx;

double mvy;

double max;

double may;

sprp = Indexing\_GetIndice (objects, spr);

fixedp = Indexing\_GetIndice (objects, fixed);

movingp = Indexing\_GetIndice (objects, moving);

getObjectValues (fixedp, &fvx, &fvy, &fax, &fay);

getObjectValues (movingp, &mvx, &mvy, &max, &may);

/\* gdbif.sleepSpin ; \*/

if (trace)

{

libc\_printf ((char \*) "fvx, fvy = %g, %g ", 20, fvx, fvy);

libc\_printf ((char \*) "fax, fay = %g, %g\\n", 19, fvx, fvy);

libc\_printf ((char \*) "mvx, mvy = %g, %g ", 20, mvx, mvy);

libc\_printf ((char \*) "max, may = %g, %g\\n", 19, mvx, mvy);

libc\_printf ((char \*) "starting with force on moving object %d = [%g, %g]\\n", 52, movingp->id, movingp->forceVec.x, movingp->forceVec.y);

}

vfm = coord\_subCoord (coord\_initCoord (fvx, fvy), coord\_initCoord (mvx, mvy));

s = coord\_subCoord (getCofG (fixed), getCofG (moving));

n = coord\_normaliseCoord (s);

springval = k\*(l1-l0);

damping = d\*((coord\_dotProd (vfm, s))/(coord\_lengthCoord (s)));

f1 = coord\_scaleCoord (n, -(springval+damping));

if (trace)

{

libc\_printf ((char \*) "spring value = %g\\n", 19, springval);

libc\_printf ((char \*) "damping value = %g\\n", 20, damping);

libc\_printf ((char \*) "vector f1 = [%g, %g]\\n", 22, f1.x, f1.y);

}

movingp->forceVec = coord\_subCoord (movingp->forceVec, f1);

sprp->s.f1 = coord\_negateCoord (f1);

sprp->s.f2 = f1;

if (trace)

{

libc\_printf ((char \*) "overall force on moving object %d = [%g, %g]\\n", 46, movingp->id, movingp->forceVec.x, movingp->forceVec.y);

factor = (sqr (d))-((4.0\*(twoDsim\_get\_mass (movingp->id)))\*k);

if (roots\_nearZero (factor))

{

libc\_printf ((char \*) "sprung object %d critical damping (increase damping or reduce spring strength of spring %d)\\n", 93, movingp->id, spr);

}

else if (factor > 0.0)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "sprung object %d is safe as it is over damped\\n", 47, movingp->id);

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "sprung object %d under damped (increase damping or reduce spring strength of spring %d)\\n", 89, movingp->id, spr);

}

}

}

/\*

calcSpringMoving - calculate the forces on, moving objects o1 and o2 attached to

spring, spr.

The spring has properties of, k, d, l0 and l1.

\*/

static void calcSpringMoving (double k, double d, double l0, double l1, unsigned int spr, unsigned int o1, unsigned int o2)

{

if (l1 > l0)

{

doCalcSpringMoving (k, d, l0, l1, spr, o1, o2);

}

}

/\*

doCalcSpringMoving - calculate the forces on, moving objects o1 and o2 attached to

spring, spr.

The spring has properties of, k, d, l0 and l1.

\*/

static void doCalcSpringMoving (double k, double d, double l0, double l1, unsigned int spr, unsigned int o1, unsigned int o2)

{

Object sprp;

Object o1p;

Object o2p;

coord\_Coord f1;

coord\_Coord vfm;

coord\_Coord fvec;

coord\_Coord s;

coord\_Coord n;

double springval;

double damping;

double fax;

double fay;

double fvx;

double fvy;

double mvx;

double mvy;

double max;

double may;

sprp = Indexing\_GetIndice (objects, spr);

o1p = Indexing\_GetIndice (objects, o1);

o2p = Indexing\_GetIndice (objects, o2);

getObjectValues (o1p, &fvx, &fvy, &fax, &fay);

getObjectValues (o2p, &mvx, &mvy, &max, &may);

/\* gdbif.sleepSpin ; \*/

if (trace)

{

libc\_printf ((char \*) "fvx, fvy = %g, %g ", 20, fvx, fvy);

libc\_printf ((char \*) "fax, fay = %g, %g\\n", 19, fvx, fvy);

libc\_printf ((char \*) "mvx, mvy = %g, %g ", 20, mvx, mvy);

libc\_printf ((char \*) "max, may = %g, %g\\n", 19, mvx, mvy);

}

vfm = coord\_subCoord (coord\_initCoord (fvx, fvy), coord\_initCoord (mvx, mvy));

s = coord\_subCoord (getCofG (o1), getCofG (o2));

n = coord\_normaliseCoord (s);

springval = k\*(l1-l0);

damping = d\*((coord\_dotProd (vfm, s))/(coord\_lengthCoord (s)));

f1 = coord\_scaleCoord (n, -(springval+damping));

if (trace)

{

libc\_printf ((char \*) "spring value = %g\\n", 19, springval);

libc\_printf ((char \*) "damping value = %g\\n", 20, damping);

libc\_printf ((char \*) "vector f1 = [%g, %g]\\n", 22, f1.x, f1.y);

}

o1p->forceVec = coord\_addCoord (o1p->forceVec, f1);

o2p->forceVec = coord\_subCoord (o2p->forceVec, f1);

sprp->s.f1 = f1;

sprp->s.f2 = coord\_negateCoord (f1);

}

/\*

doCalcSpringForce -

\*/

static void doCalcSpringForce (unsigned int id, Object idp)

{

unsigned int id1;

unsigned int id2;

switch (idp->object)

{

case springOb:

id1 = idp->s.id1;

id2 = idp->s.id2;

/\* calculate actual length of spring now. \*/

if ((twoDsim\_isFixed (id1)) && (! (twoDsim\_isFixed (id2))))

{

calcSpringFixed (idp->s.k, idp->s.d, idp->s.l0, idp->s.l1, id, id1, id2);

}

else if ((twoDsim\_isFixed (id2)) && (! (twoDsim\_isFixed (id1))))

{

/\* avoid dangling else. \*/

calcSpringFixed (idp->s.k, idp->s.d, idp->s.l0, idp->s.l1, id, id2, id1);

}

else if ((! (twoDsim\_isFixed (id1))) && (! (twoDsim\_isFixed (id2))))

{

/\* avoid dangling else. \*/

calcSpringMoving (idp->s.k, idp->s.d, idp->s.l0, idp->s.l1, id, id2, id1);

}

break;

default:

break;

}

}

/\*

calcSpringForce - calculate the forces a spring, id, has on its components.

\*/

static void calcSpringForce (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

if (objectExists (idp))

{

doCalcSpringForce (id, idp);

}

}

/\*

zeroForceEnergy - assign force vector, potential energy and kinetic energy

to zero for all objects

\*/

static void zeroForceEnergy (void)

{

unsigned int n;

unsigned int i;

Object iptr;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

iptr->forceVec = coord\_initCoord (0.0, 0.0);

iptr->ke = 0.0;

iptr->pe = 0.0;

}

i += 1;

}

}

/\*

applyDrag - apply drag to object, id, which has a spring

generated acceleration, a. We only apply the

drag if the is an acceleration (or force).

No drag is imposed if the spring is at rest.

\*/

static void applyDrag (unsigned int id, coord\_Coord a)

{

Object o;

if (! (twoDsim\_isFixed (id)))

{

o = Indexing\_GetIndice (objects, id);

if (! (roots\_nearZero (a.x)))

{

o->vx = inElastic (o, o->vx);

}

if (! (roots\_nearZero (a.y)))

{

o->vy = inElastic (o, o->vy);

}

}

}

/\*

doApplySpringForce -

\*/

static coord\_Coord doApplySpringForce (unsigned int id, coord\_Coord force)

{

coord\_Coord a;

if (twoDsim\_isFixed (id))

{

return coord\_initCoord (0.0, 0.0);

}

else

{

if (roots\_nearZero (twoDsim\_get\_mass (id)))

{

libc\_printf ((char \*) "object %d must be given a mass\\n", 32, id);

libc\_exit (1);

}

else

{

a = coord\_scaleCoord (force, 1.0/(twoDsim\_get\_mass (id)));

a = coord\_scaleCoord (a, ElasticitySpring);

applyDrag (id, a);

return a;

}

}

ReturnException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\*

doApplyForce -

\*/

static void doApplyForce (unsigned int i, Object iptr)

{

if (((isCircle (i)) || (isPolygon (i))) && (! (twoDsim\_isFixed (i))))

{

/\* avoid dangling else. \*/

if (roots\_nearZero (twoDsim\_get\_mass (i)))

{

/\* avoid dangling else. \*/

/\* suppress warning if not running. \*/

if (startedRunning)

{

libc\_printf ((char \*) "moving object %d needs a mass before a force can be applied\\n", 61, i);

}

}

else

{

if (trace)

{

libc\_printf ((char \*) "object %d has force vector (%g, %g)\\n", 38, i, iptr->forceVec.x, iptr->forceVec.y);

libc\_printf ((char \*) "object %d has (saccel = (%g, %g)\\n", 34, i, iptr->saccel.x, iptr->saccel.y);

}

iptr->saccel = coord\_initCoord (iptr->forceVec.x/(twoDsim\_get\_mass (i)), iptr->forceVec.y/(twoDsim\_get\_mass (i)));

iptr->saccel = coord\_scaleCoord (iptr->saccel, ElasticitySpring);

if (trace)

{

libc\_printf ((char \*) "object %d now has (saccel (%g, %g)\\n", 36, i, iptr->saccel.x, iptr->saccel.y);

libc\_printf ((char \*) "object %d has normal acceleration of (%g, %g)\\n", 47, i, iptr->ax, iptr->ay);

libc\_printf ((char \*) " total acceleration of (%g, %g)\\n", 46, i, iptr->ax+iptr->saccel.x, iptr->ay+iptr->saccel.y);

}

/\* iptr^.stationary := NOT (nearZero (iptr^.saccel.x) AND nearZero (iptr^.saccel.y)) \*/

}

}

else if (isSpringObject (i))

{

/\* avoid dangling else. \*/

/\* work out the acceleration due to the spring on each attached object. \*/

iptr->s.a1 = doApplySpringForce (iptr->s.id1, iptr->s.f1);

iptr->s.a2 = doApplySpringForce (iptr->s.id2, iptr->s.f2);

}

}

/\*

applyForce - translate the force into acceleration

and update stationary boolean.

\*/

static void applyForce (void)

{

unsigned int n;

unsigned int i;

Object iptr;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

iptr = Indexing\_GetIndice (objects, i);

if (objectExists (iptr))

{

doApplyForce (i, iptr);

}

i += 1;

}

}

/\*

calcSpringEnergy -

\*/

static void calcSpringEnergy (unsigned int i)

{

double M;

Object id1ptr;

Object id2ptr;

Object iptr;

Assert (isSpringObject (i), 2394);

iptr = Indexing\_GetIndice (objects, i);

iptr->ke = 0.0; /\* no kinetic energy as a spring has no mass. \*/

iptr->pe = (iptr->s.k\*(sqr (iptr->s.l0-iptr->s.l1)))/2.0; /\* no kinetic energy as a spring has no mass. \*/

id1ptr = Indexing\_GetIndice (objects, iptr->s.id1);

id2ptr = Indexing\_GetIndice (objects, iptr->s.id2);

/\* give this to the two objects attached to the spring. \*/

if (id1ptr->fixed && ! id2ptr->fixed)

{

/\* give it all to id2. \*/

id2ptr->pe = id2ptr->pe+iptr->pe;

}

else if (id2ptr->fixed && ! id1ptr->fixed)

{

/\* avoid dangling else. \*/

/\* give it all to id1. \*/

id1ptr->pe = id1ptr->pe+iptr->pe;

}

else if (! id2ptr->fixed && ! id1ptr->fixed)

{

/\* avoid dangling else. \*/

/\* give it to both id1 and id2 using their mass as a radio. \*/

M = (twoDsim\_get\_mass (iptr->s.id1))+(twoDsim\_get\_mass (iptr->s.id2));

id1ptr->pe = id1ptr->pe+((iptr->pe\*(twoDsim\_get\_mass (iptr->s.id1)))/M);

id2ptr->pe = id2ptr->pe+((iptr->pe\*(twoDsim\_get\_mass (iptr->s.id2)))/M);

}

}

/\*

calcObjectEnergy -

\*/

static void calcObjectEnergy (unsigned int i)

{

if (isSpringObject (i))

{

calcSpringEnergy (i);

}

}

/\*

calcEnergy -

\*/

static void calcEnergy (void)

{

unsigned int n;

unsigned int i;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

calcObjectEnergy (i);

i += 1;

}

}

/\*

recalculateForceEnergy - recalculate all forces and energy

for all objects.

\*/

static void recalculateForceEnergy (void)

{

if (trace)

{

libc\_printf ((char \*) "enter recalculateForceEnergy\\n", 30);

}

zeroForceEnergy ();

if (trace)

{

twoDsim\_dumpWorld ();

}

calcForce ();

/\* calcEnergy ; \*/

applyForce ();

if (trace)

{

twoDsim\_dumpWorld ();

libc\_printf ((char \*) "exit recalculateForceEnergy\\n", 29);

}

}

/\*

calcForce - calculate all forces for objects attached by springs.

\*/

static void calcForce (void)

{

unsigned int n;

unsigned int i;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

calcSpringForce (i);

i += 1;

}

}

/\*

calculateCofG -

\*/

static coord\_Coord calculateCofG (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high)

{

double A;

double B;

double C;

double D;

double a;

double x;

double y;

unsigned int i;

unsigned int j;

coord\_Coord p[\_p\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (p, p\_, (\_p\_high+1) \* sizeof (coord\_Coord));

a = (calcArea (n, (coord\_Coord \*) p, \_p\_high))\*6.0;

x = 0.0;

y = 0.0;

if (Debugging)

{

libc\_printf ((char \*) "calculateCofG begin: %d points\\n", 33, n);

for (i=0; i<=n-1; i++)

{

libc\_printf ((char \*) "%d: %g, %g\\n", 14, i, p[i].x, p[i].y);

}

}

for (i=0; i<=n-1; i++)

{

j = (i+1) % n;

if (Debugging)

{

libc\_printf ((char \*) "x = %g, y = %g\\n", 17, x, y);

A = p[i].x+p[j].x;

B = (p[i].x\*p[j].y)-(p[j].x\*p[i].y);

C = p[i].y+p[j].y;

D = (p[i].x\*p[j].y)-(p[j].x\*p[i].y);

libc\_printf ((char \*) "A = %g, B = %g\\n", 17, A, B);

libc\_printf ((char \*) "C = %g, D = %g\\n", 17, A, B);

libc\_printf ((char \*) "A \* B = %g, C \* D = %g\\n", 25, A\*B, C\*D);

}

x = x+((p[i].x+p[j].x)\*((p[i].x\*p[j].y)-(p[j].x\*p[i].y)));

y = y+((p[i].y+p[j].y)\*((p[i].x\*p[j].y)-(p[j].x\*p[i].y)));

}

if (Debugging)

{

libc\_printf ((char \*) "cofg = %g, %g\\n", 15, x/a, y/a);

}

return coord\_initCoord (x/a, y/a);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

calcArea -

\*/

static double calcArea (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high)

{

unsigned int i;

unsigned int j;

double a;

double r;

double b;

coord\_Coord p[\_p\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (p, p\_, (\_p\_high+1) \* sizeof (coord\_Coord));

a = 0.0;

if (Debugging)

{

libc\_printf ((char \*) "calculating area: ", 18);

}

for (i=0; i<=n-1; i++)

{

if (Debugging)

{

libc\_printf ((char \*) "(%g, %g) ", 9, p[i].x, p[i].y);

}

j = (i+1) % n;

r = p[i].x\*p[j].y;

b = p[i].y\*p[j].x;

if (Debugging)

{

libc\_printf ((char \*) " [x1 x y1 = %g x %g = %g = r] ", 30, p[i].x, p[j].y, r);

libc\_printf ((char \*) " [x1 x y1 = %g x %g = %g = b] ", 30, p[i].y, p[j].x, b);

}

a = (a+r)-b;

if (Debugging)

{

libc\_printf ((char \*) " [a = %g] ", 10, a);

}

}

if (Debugging)

{

libc\_printf ((char \*) "end area = %g\\n", 15, a/2.0);

}

return a/2.0;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

debugCircle - displays a circle at position, p, with radius, r, in colour, c.

\*/

static void debugCircle (coord\_Coord p, double r, deviceIf\_Colour c)

{

deviceIf\_glyphCircle (c2p (p), TRUE, Fractions\_zero (), Fractions\_putReal (r), c);

}

/\*

debugLine - displays a line from, p1, to, p2, in the debugging colour.

\*/

static void debugLine (coord\_Coord p1, coord\_Coord p2, deviceIf\_Colour c)

{

typedef struct \_T7\_a \_T7;

struct \_T7\_a { Points\_Point array[3+1]; };

\_T7 p;

coord\_Coord v;

coord\_Coord n1;

coord\_Coord n2;

sortLine (&p1, &p2);

v = coord\_subCoord (p2, p1);

coord\_perpendiculars (v, &n1, &n2);

n1 = coord\_scaleCoord (coord\_normaliseCoord (n1), thickness);

n2 = coord\_scaleCoord (coord\_normaliseCoord (n2), thickness);

p.array[0] = c2p (coord\_addCoord (p1, n1));

p.array[1] = c2p (coord\_addCoord (p2, n1));

p.array[2] = c2p (coord\_addCoord (p2, n2));

p.array[3] = c2p (coord\_addCoord (p1, n2));

deviceIf\_glyphPolygon (4, (Points\_Point \*) &p.array[0], 3, TRUE, Fractions\_zero (), c);

}

/\*

doCircle - pass parameters to the groffDevice.

\*/

static void doCircle (coord\_Coord p, double r, deviceIf\_Colour c)

{

deviceIf\_glyphCircle (c2p (p), TRUE, Fractions\_zero (), Fractions\_putReal (r), c);

}

/\*

doPolygon -

\*/

static void doPolygon (unsigned int n, coord\_Coord \*p\_, unsigned int \_p\_high, deviceIf\_Colour c)

{

typedef struct \_T8\_a \_T8;

struct \_T8\_a { Points\_Point array[MaxPolygonPoints+1]; };

\_T8 points;

unsigned int i;

coord\_Coord p[\_p\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (p, p\_, (\_p\_high+1) \* sizeof (coord\_Coord));

for (i=0; i<=n-1; i++)

{

if (Debugging)

{

libc\_printf ((char \*) "polygon point %d: %g, %g\\n", 26, i, p[i].x, p[i].y);

}

points.array[i] = c2p ((coord\_Coord) p[i]);

}

deviceIf\_glyphPolygon (n, (Points\_Point \*) &points.array[0], MaxPolygonPoints, TRUE, Fractions\_zero (), c);

}

/\*

drawBoarder -

\*/

static void drawBoarder (deviceIf\_Colour c)

{

typedef struct \_T9\_a \_T9;

struct \_T9\_a { Points\_Point array[3+1]; };

\_T9 p;

p.array[0] = Points\_initPoint (Fractions\_zero (), Fractions\_zero ());

p.array[1] = Points\_initPoint (Fractions\_one (), Fractions\_zero ());

p.array[2] = Points\_initPoint (Fractions\_one (), Fractions\_one ());

p.array[3] = Points\_initPoint (Fractions\_zero (), Fractions\_one ());

deviceIf\_glyphPolygon (4, (Points\_Point \*) &p.array[0], 3, FALSE, Fractions\_initFract (0, 1, 100), c);

}

/\*

drawBackground -

\*/

static void drawBackground (deviceIf\_Colour c)

{

typedef struct \_T10\_a \_T10;

struct \_T10\_a { Points\_Point array[3+1]; };

\_T10 p;

p.array[0] = Points\_initPoint (Fractions\_zero (), Fractions\_zero ());

p.array[1] = Points\_initPoint (Fractions\_one (), Fractions\_zero ());

p.array[2] = Points\_initPoint (Fractions\_one (), Fractions\_one ());

p.array[3] = Points\_initPoint (Fractions\_zero (), Fractions\_one ());

deviceIf\_glyphPolygon (4, (Points\_Point \*) &p.array[0], 3, TRUE, Fractions\_zero (), c);

}

/\*

getVelCoord - returns a velocity coordinate pair for Object, o.

\*/

static coord\_Coord getVelCoord (Object o)

{

checkDeleted (o);

if (o->fixed || o->stationary)

{

return coord\_initCoord (0.0, 0.0);

}

else

{

return coord\_initCoord (o->vx, o->vy);

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getAccelCoord - returns an acceleration coordinate pair for Object, o.

\*/

static coord\_Coord getAccelCoord (Object o)

{

checkDeleted (o);

if (o->fixed || o->stationary)

{

return coord\_initCoord (0.0, 0.0);

}

else

{

return coord\_addCoord (coord\_initCoord (o->ax, (o->ay+simulatedGravity)+o->gravity), o->saccel);

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doCircleFrame -

\*/

static void doCircleFrame (Object optr, double dt, deviceIf\_Colour col)

{

coord\_Coord vc;

coord\_Coord ac;

vc = getVelCoord (optr);

ac = getAccelCoord (optr);

doCircle (newPositionCoord (optr->c.pos, vc, ac, dt), optr->c.r, col);

}

/\*

doPolygonFrame -

\*/

static void doPolygonFrame (Object optr, double dt, deviceIf\_Colour col)

{

typedef struct \_T11\_a \_T11;

struct \_T11\_a { coord\_Coord array[MaxPolygonPoints+1]; };

unsigned int i;

\_T11 po;

coord\_Coord co;

coord\_Coord vc;

coord\_Coord ac;

vc = getVelCoord (optr);

ac = getAccelCoord (optr);

for (i=0; i<=optr->p.nPoints-1; i++)

{

po.array[i] = newPositionRotationCoord (optr->p.cOfG, vc, ac, dt, optr->angularVelocity, optr->angleOrientation, (polar\_Polar) optr->p.points.array[i]);

if (Debugging)

{

libc\_printf ((char \*) "po[%d].x = %g, po[%d].y = %g\\n", 30, i, po.array[i].x, i, po.array[i].y);

}

co = coord\_addCoord (optr->p.cOfG, polar\_polarToCoord (polar\_rotatePolar ((polar\_Polar) optr->p.points.array[i], optr->angleOrientation)));

if (Debugging)

{

libc\_printf ((char \*) " [co.x = %g, co.y = %g]\\n", 25, co.x, co.y);

}

if (roots\_nearZero (dt))

{

if ((! (roots\_nearZero (co.x-po.array[i].x))) || (! (roots\_nearZero (co.y-po.array[i].y))))

{

libc\_printf ((char \*) "these values should be the same\\n", 33);

libc\_exit (1);

}

}

}

doPolygon (optr->p.nPoints, (coord\_Coord \*) &po.array[0], MaxPolygonPoints, col);

}

/\*

doSpringFrame -

\*/

static void doSpringFrame (Object optr, double dt, unsigned int col)

{

typedef struct \_T12\_a \_T12;

struct \_T12\_a { coord\_Coord array[3+1]; };

unsigned int i;

coord\_Coord p;

coord\_Coord s1;

coord\_Coord s2;

Object o1;

Object o2;

\_T12 po;

coord\_Coord co;

coord\_Coord vc;

coord\_Coord ac;

double w2;

Assert (optr->object == springOb, 3000);

if (optr->s.draw)

{

w2 = optr->s.width/2.0;

o1 = Indexing\_GetIndice (objects, optr->s.id1);

o2 = Indexing\_GetIndice (objects, optr->s.id2);

s1 = newPositionCoord (getCofG (optr->s.id1), getVelCoord (o1), getAccelCoord (o1), dt);

s2 = newPositionCoord (getCofG (optr->s.id2), getVelCoord (o2), getAccelCoord (o2), dt);

p = coord\_scaleCoord (coord\_normaliseCoord (coord\_perpendicular (coord\_subCoord (s1, s2))), w2);

po.array[0] = coord\_addCoord (s1, p);

po.array[1] = coord\_subCoord (s1, p);

po.array[2] = coord\_subCoord (s2, p);

po.array[3] = coord\_addCoord (s2, p);

doPolygon (4, (coord\_Coord \*) &po.array[0], 3, (deviceIf\_Colour) optr->s.drawColour);

}

}

/\*

doDrawFrame -

\*/

static void doDrawFrame (Object optr, double dt, deviceIf\_Colour col)

{

coord\_Coord ac;

coord\_Coord vc;

if (DebugTrace)

{

libc\_printf ((char \*) "doDrawFrame (%g)\\n", 18, dt);

}

checkDeleted (optr);

switch (optr->object)

{

case circleOb:

doCircleFrame (optr, dt, col);

break;

case springOb:

doSpringFrame (optr, dt, (unsigned int) col);

break;

case polygonOb:

doPolygonFrame (optr, dt, col);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

getEventObjects -

\*/

static void getEventObjects (Object \*id1, Object \*id2, eventQueue e)

{

Object id;

(\*id1) = NULL;

(\*id2) = NULL;

if (e != NULL)

{

switch (e->ePtr->etype)

{

case circlesEvent:

(\*id1) = Indexing\_GetIndice (objects, e->ePtr->cc.cid1);

(\*id2) = Indexing\_GetIndice (objects, e->ePtr->cc.cid2);

break;

case circlePolygonEvent:

(\*id1) = Indexing\_GetIndice (objects, e->ePtr->cp.cid);

(\*id2) = Indexing\_GetIndice (objects, e->ePtr->cp.pid);

break;

case polygonPolygonEvent:

(\*id1) = Indexing\_GetIndice (objects, e->ePtr->pp.pid1);

(\*id2) = Indexing\_GetIndice (objects, e->ePtr->pp.pid2);

break;

case springEvent:

id = Indexing\_GetIndice (objects, e->ePtr->sp.id);

(\*id1) = Indexing\_GetIndice (objects, id->s.id1);

(\*id2) = Indexing\_GetIndice (objects, id->s.id2);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

}

/\*

getColour -

\*/

static deviceIf\_Colour getColour (Object optr)

{

switch (optr->object)

{

case polygonOb:

return optr->p.col;

break;

case circleOb:

return optr->c.col;

break;

case springOb:

return deviceIf\_white ();

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getSpringColour -

\*/

static deviceIf\_Colour getSpringColour (void)

{

if (haveSpringColour)

{

return springColour;

}

else

{

return deviceIf\_red ();

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getCollisionColour -

\*/

static deviceIf\_Colour getCollisionColour (void)

{

if (haveCollisionColour)

{

return collisionColour;

}

else

{

return deviceIf\_blue ();

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getEventObjectColour -

\*/

static deviceIf\_Colour getEventObjectColour (eventQueue e, Object optr)

{

Object id1;

Object id2;

getEventObjects (&id1, &id2, e);

if ((e == NULL) || ((id1 != optr) && (id2 != optr)))

{

return getColour (optr);

}

else

{

switch (e->kind)

{

case collisionKind:

return getCollisionColour ();

break;

case springKind:

return getSpringColour ();

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

equalBroadphase - return TRUE if the pair of objects in, b, is the same

as (i, j).

\*/

static unsigned int equalBroadphase (broadphase b, unsigned int i, unsigned int j)

{

return (b->o0 == i) && (b->o1 == j);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

assignBroadphase - assign each field in broadphase to: i, j and next.

b is returned.

\*/

static broadphase assignBroadphase (broadphase b, unsigned int i, unsigned int j, broadphase next)

{

b->o0 = i;

b->o1 = j;

b->next = next;

return b;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newBroadphase - return a new initialised broadphase element.

\*/

static broadphase newBroadphase (unsigned int i, unsigned int j, broadphase next)

{

broadphase b;

if (freeBroadphase == NULL)

{

Storage\_ALLOCATE ((void \*\*) &b, sizeof (\_T5));

}

else

{

b = freeBroadphase;

freeBroadphase = freeBroadphase->next;

}

return assignBroadphase (b, i, j, next);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

addBroadphase - adds, pair, i, j, onto the head of the broadphase list

as long as the pair is unique. It returns the new element

which is chained to head.

\*/

static broadphase addBroadphase (broadphase head, unsigned int i, unsigned int j)

{

broadphase b;

if (head == NULL)

{

return newBroadphase (i, j, (broadphase) NULL);

}

else

{

b = head;

while (b != NULL)

{

if ((equalBroadphase (b, i, j)) || (equalBroadphase (b, j, i)))

{

/\* already seen this pair, therefore return head. \*/

return head;

}

b = b->next;

}

return newBroadphase (i, j, head);

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

initBroadphase - the constructor which returns a new broadphase list of objects.

\*/

static broadphase initBroadphase (void)

{

broadphase head;

unsigned int i;

unsigned int j;

unsigned int n;

head = NULL;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

j = 1;

while (j <= n)

{

if ((((i != j) && (objectIdExists (i))) && (objectIdExists (j))) && ((! (twoDsim\_isFixed (i))) || (! (twoDsim\_isFixed (j)))))

{

/\* either i or j is moving therefore there might be interpenetration. \*/

head = addBroadphase (head, i, j);

}

j += 1;

}

i += 1;

}

return head;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

killBroadphase - returns list, head, back to the freeBroadphase list.

\*/

static void killBroadphase (broadphase \*head)

{

broadphase last;

if ((\*head) != NULL)

{

if (freeBroadphase == NULL)

{

freeBroadphase = (\*head);

}

else

{

last = (\*head);

while (last->next != NULL)

{

last = last->next;

}

last->next = freeBroadphase;

freeBroadphase = (\*head);

}

(\*head) = NULL;

}

}

/\*

checkFrameBasedInterpen -

\*/

static void checkFrameBasedInterpen (void)

{

framePolygons = determineFrameBased ();

if (FrameSprings || framePolygons)

{

/\* are we using frame based simulation to solve spring motion. \*/

updatePhysics (TRUE);

}

if (framePolygons)

{

checkFrameInterpen ();

}

if (PolygonDebugging)

{

printQueue ();

}

}

/\*

drawFrame - draws the current world into the frame buffer.

If e is not NIL then it will be a collision event

which describes the objects colliding. The

drawFrame will draw these objects using

the debugging colour.

\*/

static void drawFrame (eventQueue e)

{

double dt;

unsigned int i;

unsigned int n;

Object optr;

Assert (((e == NULL) || (e->kind == collisionKind)) || (e->kind == springKind), 3315);

if (DebugTrace)

{

libc\_printf ((char \*) "start drawFrame\\n", 17);

}

if (writeTimeDelay)

{

deviceIf\_writeTime (currentTime-lastDrawTime);

}

lastDrawTime = currentTime;

dt = currentTime-lastUpdateTime;

if (DebugTrace)

{

libc\_printf ((char \*) "before drawBoarder\\n", 20);

}

drawBoarder ((deviceIf\_Colour) deviceIf\_black ());

if (DebugTrace)

{

libc\_printf ((char \*) "after drawBoarder\\n", 19);

}

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

optr = Indexing\_GetIndice (objects, i);

if ((optr != NULL) && (! optr->deleted) && (optr->visible))

{

if (Debugging)

{

dumpObject (optr);

}

/\* printf ("before doDrawFrame

"); \*/

doDrawFrame (optr, dt, getEventObjectColour (e, optr));

}

/\* printf ("after doDrawFrame

"); \*/

i += 1;

}

}

/\*

drawFrameEvent -

\*/

static void drawFrameEvent (eventQueue e)

{

if (DebugTrace)

{

libc\_printf ((char \*) "start drawFrameEvent\\n", 22);

}

checkFrameBasedInterpen ();

if (DebugTrace)

{

libc\_printf ((char \*) "before frameNote\\n", 18);

}

deviceIf\_frameNote ();

if (DebugTrace)

{

libc\_printf ((char \*) "before drawFrame\\n", 18);

}

drawFrame ((eventQueue) NULL);

if (DebugTrace)

{

libc\_printf ((char \*) "before flipBuffer\\n", 19);

}

deviceIf\_flipBuffer ();

if (DebugTrace)

{

libc\_printf ((char \*) "before addEvent\\n", 17);

}

addEvent ((eventKind) frameKind, 1.0/framesPerSecond, (eventProc) {(eventProc\_t) drawFrameEvent});

if (DebugTrace)

{

libc\_printf ((char \*) "collectAll\\n", 12);

}

GC\_collectAll ();

if (DebugTrace)

{

libc\_printf ((char \*) "end drawFrameEvent\\n", 20);

}

}

/\*

incRadians - return (a + b) mod 2pi. The value returned will be between 0..2pi

\*/

static double incRadians (double a, double b)

{

a = a+b;

if (roots\_nearZero (a))

{

a = 0.0;

}

else

{

while (a < 0.0)

{

a = a+(2.0\*MathLib0\_pi);

}

while (a > (2.0\*MathLib0\_pi))

{

a = a-(2.0\*MathLib0\_pi);

}

}

return a;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

updatePolygon -

\*/

static void updatePolygon (Object optr, double dt)

{

coord\_Coord ac;

if (! optr->deleted)

{

ac = getAccelCoord (optr);

optr->p.oldcOfG = optr->p.cOfG;

optr->p.cOfG.x = newPositionScalar (optr->p.cOfG.x, optr->vx, ac.x, dt);

optr->p.cOfG.y = newPositionScalar (optr->p.cOfG.y, optr->vy, ac.y, dt);

optr->vx = optr->vx+(optr->ax\*dt);

optr->vy = optr->vy+(ac.y\*dt);

optr->angleOrientation = incRadians (optr->angleOrientation, optr->angularVelocity\*dt);

}

}

/\*

updateCircle -

\*/

static void updateCircle (Object optr, double dt)

{

double vn;

coord\_Coord ac;

if (! optr->deleted)

{

/\* update vx and pos.x \*/

ac = getAccelCoord (optr);

optr->c.pos.x = newPositionScalar (optr->c.pos.x, optr->vx, ac.x, dt);

optr->vx = optr->vx+(ac.x\*dt);

/\* update vy and pos.y \*/

optr->c.pos.y = newPositionScalar (optr->c.pos.y, optr->vy, ac.y, dt);

optr->vy = optr->vy+(ac.y\*dt);

}

}

/\*

updateSpring - update the current length, l1, field of the spring.

\*/

static void updateSpring (Object optr, double dt)

{

optr->s.l1 = coord\_lengthCoord (coord\_subCoord (getCofG (optr->s.id1), getCofG (optr->s.id2)));

}

/\*

updateOb -

\*/

static void updateOb (Object optr, double dt)

{

if ((! optr->deleted && ! optr->fixed) && ! optr->stationary)

{

switch (optr->object)

{

case polygonOb:

updatePolygon (optr, dt);

break;

case circleOb:

updateCircle (optr, dt);

break;

case springOb:

updateSpring (optr, dt);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

}

/\*

doUpdatePhysics - updates all positions of objects based on the passing of

dt seconds.

\*/

static void doUpdatePhysics (double dt)

{

unsigned int i;

unsigned int n;

Object optr;

n = Indexing\_HighIndice (objects);

i = 1;

/\* springs are dependant on circles and polygons, so these are moved first. \*/

while (i <= n)

{

optr = Indexing\_GetIndice (objects, i);

if (! (isSpringObject (i)))

{

updateOb (optr, dt);

}

i += 1;

}

/\* now the springs. \*/

i = 1;

while (i <= n)

{

optr = Indexing\_GetIndice (objects, i);

if (isSpringObject (i))

{

updateOb (optr, dt);

}

i += 1;

}

}

/\*

updatePhysics - updates the velocity of all objects based on

the elapsed time from the last collision until now.

\*/

static void updatePhysics (unsigned int recalculateForce)

{

if (trace)

{

libc\_printf ((char \*) "enter updatePhysics\\n", 21);

}

doUpdatePhysics (currentTime-lastUpdateTime);

if (recalculateForce)

{

recalculateForceEnergy ();

}

lastUpdateTime = currentTime;

if (trace)

{

libc\_printf ((char \*) "exit updatePhysics\\n", 20);

}

}

/\*

displayEvent -

\*/

static void displayEvent (eventQueue e)

{

libc\_printf ((char \*) "%g %p ", 6, e->time\_, e->p);

if (e->kind == frameKind)

{

libc\_printf ((char \*) "frameKind ", 10);

}

else if (e->kind == collisionKind)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "collisionKind ", 14);

}

else if (e->kind == functionKind)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "functionEvent ", 14);

}

else if (e->kind == springKind)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "springEvent ", 12);

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "unknown kind ", 13);

}

if (e->ePtr == NULL)

{

libc\_printf ((char \*) "\\n", 2);

}

else

{

switch (e->ePtr->etype)

{

case frameEvent:

libc\_printf ((char \*) "display frame event\\n", 21);

break;

case circlesEvent:

libc\_printf ((char \*) "circle %d and circle %d colliding event\\n", 41, e->ePtr->cc.cid1, e->ePtr->cc.cid2);

break;

case circlePolygonEvent:

libc\_printf ((char \*) "circle %d and polygon %d colliding event\\n", 42, e->ePtr->cp.cid, e->ePtr->cp.pid);

if (e->ePtr->cp.wpid == history\_corner)

{

libc\_printf ((char \*) " hits polygon on its corner %d\\n", 33, e->ePtr->cp.pointNo);

}

else

{

libc\_printf ((char \*) " hits polygon on its edge %d\\n", 31, e->ePtr->cp.lineNo);

}

break;

case polygonPolygonEvent:

libc\_printf ((char \*) "polygon %d and polygon %d colliding event\\n", 43, e->ePtr->pp.pid1, e->ePtr->pp.pid2);

if (e->ePtr->pp.wpid1 == history\_corner)

{

libc\_printf ((char \*) " polygon (%d) corner %d hits", 29, e->ePtr->pp.pid1, e->ePtr->pp.lineCorner1);

}

else

{

libc\_printf ((char \*) " polygon (%d) edge %d hits ", 28, e->ePtr->pp.pid1, e->ePtr->pp.lineCorner1);

}

if (e->ePtr->pp.wpid2 == history\_corner)

{

libc\_printf ((char \*) " polygon (%d) on corner %d\\n", 28, e->ePtr->pp.pid2, e->ePtr->pp.lineCorner2);

}

else

{

libc\_printf ((char \*) " polygon (%d) on edge %d\\n", 26, e->ePtr->pp.pid2, e->ePtr->pp.lineCorner2);

}

break;

case functionEvent:

libc\_printf ((char \*) "function event %d (%d)\\n", 24, e->ePtr->fc.id, e->ePtr->fc.param);

break;

case springEvent:

libc\_printf ((char \*) "spring %d reached ", 18, e->ePtr->sp.id);

switch (e->ePtr->sp.type)

{

case history\_midPoint:

libc\_printf ((char \*) "midpoint", 8);

break;

case history\_endPoint:

libc\_printf ((char \*) "endpoint", 8);

break;

case history\_callPoint:

libc\_printf ((char \*) "callpoint", 9);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

libc\_printf ((char \*) "\\n", 2);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

}

/\*

printQueue - prints out the event queue.

\*/

static void printQueue (void)

{

eventQueue e;

if (DebugTrace || TRUE)

{

libc\_printf ((char \*) "current time = %g, lastDrawTime = %g, lastUpdateTime = %g\\n", 61, currentTime, lastDrawTime, lastUpdateTime);

libc\_printf ((char \*) "The event queue\\n", 17);

libc\_printf ((char \*) "===============\\n", 17);

e = eventQ;

while (e != NULL)

{

displayEvent (e);

e = e->next;

}

}

}

/\*

updateStats -

\*/

static void updateStats (double dt)

{

unsigned int lastTime;

unsigned int nextTime;

unsigned int fps;

lastTime = ((int) (currentTime\*10.0));

nextTime = ((int) ((currentTime+dt)\*10.0));

if (lastTime != nextTime)

{

fps = delay\_getActualFPS ();

libc\_printf ((char \*) "%d.%d seconds simulated, fps: %d\\n", 34, nextTime / 10, nextTime % 10, fps);

}

}

/\*

doFunctionEvent -

\*/

static void doFunctionEvent (eventQueue e)

{

/\* nothing to do. \*/

if (DebugTrace)

{

libc\_printf ((char \*) "doFunctionEvent\\n", 17);

printQueue ();

}

resetQueue ();

if (DebugTrace)

{

libc\_printf ((char \*) "adjusting the queue\\n", 21);

printQueue ();

libc\_printf ((char \*) "end of FunctionEvent\\n", 22);

}

}

/\*

doNextEvent -

\*/

static double doNextEvent (void)

{

eventQueue e;

double dt;

eventProc p;

if (eventQ == NULL)

{

libc\_printf ((char \*) "no more events on the event queue\\n", 35);

libc\_exit (1);

return 0.0;

}

else

{

if (trace)

{

libc\_printf ((char \*) "inside doNextEvent\\n", 20);

printQueue ();

}

e = eventQ;

eventQ = eventQ->next;

dt = e->time\_;

p = e->p;

currentTime = currentTime+dt;

Assert (((((p.proc == drawFrameEvent) || (p.proc == doCollision)) || (p.proc == debugFrame)) || (p.proc == doFunctionEvent)) || (p.proc == doSpring), 3734);

(\*p.proc) (e);

disposeDesc (&e->ePtr);

disposeEvent (e);

updateStats (dt);

checkMicroInterpen ();

return dt;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkZero -

\*/

static void checkZero (double \*v)

{

if ((((\*v) > 0.0) && ((\*v) < 0.01)) || (((\*v) < 0.0) && ((\*v) > -0.01)))

{

(\*v) = 0.0;

}

}

/\*

checkZeroCoord -

\*/

static coord\_Coord checkZeroCoord (coord\_Coord c)

{

if (roots\_nearZero (c.x))

{

c.x = 0.0;

}

if (roots\_nearZero (c.y))

{

c.y = 0.0;

}

return c;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

inElasticSpring -

\*/

static void inElasticSpring (double \*v)

{

(\*v) = (\*v)\*ElasticitySpring;

checkZero (v);

}

/\*

inElastic -

\*/

static double inElastic (Object o, double v)

{

v = v \*o->elasticity;

checkZero (&v);

return v;

}

/\*

nearZeroVelocity - returns TRUE if, r, is close to 0.0

\*/

static unsigned int nearZeroVelocity (double r)

{

if (r >= 0.0)

{

return r < 0.01;

}

else

{

return -r < 0.01;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkStationary - checks to see if object, o, should be put into

the stationary state.

\*/

static void checkStationary (Object o)

{

if (objectExists (o))

{

if (! o->fixed)

{

o->vx = inElastic (o, o->vx);

o->vy = inElastic (o, o->vy);

}

/\*

stationary := nearZeroVelocity (vx) AND nearZeroVelocity (vy) AND

nearZero (ax) AND nearZero (ay) ;

IF stationary

THEN

vx := 0.0 ;

vy := 0.0 ;

IF Debugging

THEN

dumpObject (o)

END

END

\*/

}

}

/\*

checkStationarySpring - checks to see if object, o, should be put into

the stationary state.

\*/

static void checkStationarySpring (Object o)

{

if (objectExists (o))

{

if (! o->fixed)

{

o->forceVec = coord\_scaleCoord (o->forceVec, ElasticitySpring);

inElasticSpring (&o->vx);

inElasticSpring (&o->vy);

o->stationary = (nearZeroVelocity (o->vx)) && (nearZeroVelocity (o->vy));

if (o->stationary)

{

o->vx = 0.0;

o->vy = 0.0;

if (Debugging)

{

dumpObject (o);

}

}

}

}

}

/\*

checkStationaryCollision - stationary object, a, has been bumped by

moving object, b. We move a slightly and

give it an initial velocity and change its

state from stationary to moving.

\*/

static void checkStationaryCollision (Object a, Object b)

{

if (a->stationary && ! a->deleted)

{

/\* avoid dangling else. \*/

if (Debugging)

{

libc\_printf ((char \*) "object %d has bumped into a stationary object %d\\n", 50, b->id, a->id);

}

/\* gdbif.sleepSpin ; \*/

a->vy = 1.0;

if (a->c.pos.x < b->c.pos.x)

{

a->c.pos.x = a->c.pos.x-0.001;

}

else

{

a->c.pos.x = a->c.pos.x+0.001;

}

a->c.pos.y = a->c.pos.y+0.001;

a->stationary = FALSE;

if (Debugging)

{

dumpObject (a);

}

}

else if (b->stationary && ! b->deleted)

{

/\* avoid dangling else. \*/

checkStationaryCollision (b, a);

}

}

/\*

collideFixedCircles - works out the new velocity given that the circle

movable collides with the fixed circle.

\*/

static void collideFixedCircles (Object movable, Object fixed)

{

collideAgainstFixedCircle (movable, fixed->c.pos);

}

/\*

collideAgainstFixedCircle - the movable object collides against a point, center.

center, is the center point of the other fixed circle.

This procedure works out the new velocity of the movable

circle given these constraints.

\*/

static void collideAgainstFixedCircle (Object movable, coord\_Coord center)

{

double r;

double j;

coord\_Coord c;

coord\_Coord normalCollision;

coord\_Coord relativeVelocity;

/\* calculate normal collision value \*/

c.x = movable->c.pos.x-center.x;

c.y = movable->c.pos.y-center.y;

r = libm\_sqrt ((c.x\*c.x)+(c.y\*c.y));

normalCollision.x = c.x/r;

normalCollision.y = c.y/r;

relativeVelocity.x = movable->vx;

relativeVelocity.y = movable->vy;

j = (-((1.0+1.0)\*((relativeVelocity.x\*normalCollision.x)+(relativeVelocity.y\*normalCollision.y))))/(((normalCollision.x\*normalCollision.x)+(normalCollision.y\*normalCollision.y))\*(1.0/movable->c.mass));

movable->vx = movable->vx+((j\*normalCollision.x)/movable->c.mass);

movable->vy = movable->vy+((j\*normalCollision.y)/movable->c.mass);

checkStationary (movable);

}

/\*

collideMovableCircles -

\*/

static void collideMovableCircles (Object iptr, Object jptr)

{

double r;

double j;

coord\_Coord c;

coord\_Coord normalCollision;

coord\_Coord relativeVelocity;

/\* calculate normal collision value \*/

c.x = iptr->c.pos.x-jptr->c.pos.x;

c.y = iptr->c.pos.y-jptr->c.pos.y;

r = libm\_sqrt ((c.x\*c.x)+(c.y\*c.y));

normalCollision.x = c.x/r;

normalCollision.y = c.y/r;

relativeVelocity.x = iptr->vx-jptr->vx;

relativeVelocity.y = iptr->vy-jptr->vy;

j = (-((1.0+1.0)\*((relativeVelocity.x\*normalCollision.x)+(relativeVelocity.y\*normalCollision.y))))/(((normalCollision.x\*normalCollision.x)+(normalCollision.y\*normalCollision.y))\*((1.0/iptr->c.mass)+(1.0/jptr->c.mass)));

iptr->vx = iptr->vx+((j\*normalCollision.x)/iptr->c.mass);

iptr->vy = iptr->vy+((j\*normalCollision.y)/iptr->c.mass);

jptr->vx = jptr->vx-((j\*normalCollision.x)/jptr->c.mass);

jptr->vy = jptr->vy-((j\*normalCollision.y)/jptr->c.mass);

checkStationaryCollision (iptr, jptr);

checkStationary (iptr);

checkStationary (jptr);

}

/\*

circleCollision - call fixed or movable circle collision depending upon whether

one or two circles are fixed.

Apart from taking into account rotation of either circle this

is complete.

\*/

static void circleCollision (Object iptr, Object jptr)

{

if (iptr->fixed)

{

collideFixedCircles (jptr, iptr);

}

else if (jptr->fixed)

{

/\* avoid dangling else. \*/

collideFixedCircles (iptr, jptr);

}

else

{

/\* avoid dangling else. \*/

collideMovableCircles (iptr, jptr);

}

}

/\*

collideCircleAgainstFixedEdge - modifies the circle velocity based upon the edge it hits.

We use the formula:

V = 2 \* (-I . N ) \* N + I

where:

I is the initial velocity vector

V is the final velocity vector

N is the normal to the line

\*/

static void collideCircleAgainstFixedEdge (Object cPtr, coord\_Coord p1, coord\_Coord p2)

{

coord\_Coord l;

coord\_Coord vel;

/\* firstly we need to find the normal to the line \*/

sortLine (&p1, &p2); /\* p1 and p2 are the start end positions of the line \*/

l = coord\_subCoord (p2, p1); /\* l is the vector p1 -> p2 \*/

vel = coord\_initCoord (cPtr->vx, cPtr->vy); /\* vel is the initial velocity \*/

vel = reflect (vel, l); /\* vel is the initial velocity \*/

cPtr->vx = vel.x; /\* update velocity of object, cPtr \*/

cPtr->vy = vel.y; /\* update velocity of object, cPtr \*/

checkStationary (cPtr);

}

/\*

circlePolygonCollision -

\*/

static void circlePolygonCollision (eventQueue e, Object cPtr, Object pPtr)

{

unsigned int ln;

coord\_Coord p1;

coord\_Coord p2;

if (e->ePtr->etype == circlePolygonEvent)

{

switch (e->ePtr->cp.wpid)

{

case history\_corner:

if (cPtr->fixed)

{

/\* --fixme-- to do later \*/

libc\_printf ((char \*) "moving polygon hits a fixed circle, on the polygon corner, unimplemented at present, --fixme--\\n", 96);

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

}

else if (pPtr->fixed)

{

/\* avoid dangling else. \*/

/\* moving circle hits fixed polygon corner \*/

collideAgainstFixedCircle (cPtr, e->ePtr->cp.cPoint);

}

else

{

/\* avoid dangling else. \*/

/\* both moving, to do later --fixme-- \*/

libc\_printf ((char \*) "collision between circle and polygon which are both moving, on the polygon corner, unimplemented at present, --fixme--\\n", 120);

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

}

break;

case history\_edge:

if (cPtr->fixed)

{

/\* --fixme-- to do later \*/

libc\_printf ((char \*) "collision between a fixed circle and a moving polygon, on the polygon edge, unimplemented at present, --fixme--\\n", 113);

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

}

else if (pPtr->fixed)

{

/\* avoid dangling else. \*/

/\* moving circle hits fixed polygon, on the edge \*/

ln = e->ePtr->cp.lineNo;

getPolygonLine (ln, pPtr, &p1, &p2);

collideCircleAgainstFixedEdge (cPtr, p1, p2);

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "collision between a moving circle and a moving polygon, on the polygon edge, unimplemented at present, --fixme--\\n", 114);

/\* both moving, to do later --fixme-- \*/

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

}

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

else

{

M2RTS\_HALT (-1); /\* should be circlePolygonEvent \*/

\_\_builtin\_unreachable ();

}

}

/\*

collidePolygonAgainstFixedCircle - polygon, o, is moving and has hit

a fixed circle at position, collision.

\*/

static void collidePolygonAgainstFixedCircle (Object o, coord\_Coord collision)

{

collideAgainstFixedCircle (o, collision);

if (Debugging)

{

dumpObject (o);

}

}

/\*

collidePolygonAgainstFixedEdge - p1, p2 is the fixed edge and, o, is the

moving polygon.

\*/

static void collidePolygonAgainstFixedEdge (Object o, coord\_Coord p1, coord\_Coord p2)

{

/\* find the point of collision, this is the mid point along

the shortest intersection between, o, and p1->p2. \*/

collideCircleAgainstFixedEdge (o, p1, p2);

if (Debugging)

{

dumpObject (o);

}

}

/\*

rotationalVelocity - return a vector containing the velocity

if polygon is rotating at angular speed, w,

and non rotation speed is, u and the relative

position is, r.

\*/

static coord\_Coord rotationalVelocity (double w, coord\_Coord u, coord\_Coord r)

{

return coord\_addCoord (u, coord\_scaleCoord (r, w));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

calcInertia - calculate the inertia of a polygon.

\*/

static void calcInertia (Object id)

{

coord\_Coord ci;

coord\_Coord cj;

double top;

double bot;

double t;

double b;

unsigned int i;

unsigned int j;

if (id->object == polygonOb)

{

top = 0.0;

bot = 0.0;

for (i=0; i<=id->p.nPoints-1; i++)

{

j = (i+1) % id->p.nPoints;

ci = coord\_addCoord (id->p.cOfG, polar\_polarToCoord (polar\_rotatePolar ((polar\_Polar) id->p.points.array[i], id->angleOrientation)));

cj = coord\_addCoord (id->p.cOfG, polar\_polarToCoord (polar\_rotatePolar ((polar\_Polar) id->p.points.array[j], id->angleOrientation)));

t = ((((((sqr (ci.x))+(sqr (ci.y)))+(ci.x\*cj.x))+(ci.y\*cj.y))+(sqr (cj.x)))+(sqr (cj.y)))\*((ci.x\*cj.y)-(cj.x\*ci.y));

top = top+t;

b = (ci.x\*cj.y)-(cj.x\*ci.y);

bot = bot+b;

}

id->inertia = (id->p.mass\*top)/(bot\*6.0);

}

else

{

id->inertia = id->c.mass;

}

}

/\*

updatePolygonVelocity - update both linear and angular velocities of

object, o, using impulse, j.

n is the perpendicular to the impact.

rp is the distance of the c of g of an object,

o, to the point of collision.

\*/

static void updatePolygonVelocity (Object o, double j, coord\_Coord n, coord\_Coord rpn)

{

coord\_Coord va;

/\* linear velocity update. (eq 8a) \*/

va = coord\_initCoord (o->vx, o->vy);

va = coord\_addCoord (va, coord\_scaleCoord (n, j/o->p.mass));

o->vx = va.x;

o->vy = va.y;

/\* angular velocity update. (eq 8b) \*/

o->angularVelocity = o->angularVelocity+((coord\_dotProd (rpn, coord\_scaleCoord (n, j)))/o->inertia);

}

/\*

polygonPolygonCollision - two polygons collide, we call the appropriate routines

depending upon whether one polygon is fixed.

\*/

static void polygonPolygonCollision (eventQueue e, Object id1, Object id2)

{

collidePolygonAgainstMovingPolygon (e, id1, id2);

}

/\*

reflect - reflect velocity, v, off line, l.

V = 2 \* (-I . n ) \* n + I

where:

I is the initial velocity vector

V is the final velocity vector

n is the normal to the line, l.

\*/

static coord\_Coord reflect (coord\_Coord v, coord\_Coord l)

{

coord\_Coord n;

n = coord\_perpendicular (l);

n = coord\_normaliseCoord (n);

return coord\_addCoord (coord\_scaleCoord (n, -(2.0\*(coord\_dotProd (v, n)))), v);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

collidePolygonAgainstFixedPolygon - moving, and, fixed, are two polygons.

Work out the new velocity of the moving polygon

and also rotation velocity.

\*/

static void collidePolygonAgainstFixedPolygon (eventQueue e, Object moving, Object fixed)

{

double I\_;

double m;

double j;

coord\_Coord l;

coord\_Coord n;

coord\_Coord n2;

coord\_Coord rap;

coord\_Coord rapn;

coord\_Coord p1;

coord\_Coord p2;

coord\_Coord p;

coord\_Coord v;

M2RTS\_HALT (-1); /\* this function does not work and is not used. \*/

\_\_builtin\_unreachable ();

Assert (! moving->fixed, 4320); /\* this function does not work and is not used. \*/

Assert (fixed->fixed, 4321);

Assert (e->ePtr->etype == polygonPolygonEvent, 4322);

if (Debugging)

{

libc\_printf ((char \*) "collidePolygonAgainstFixedPolygon\\n", 35);

}

drawFrame (e); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

p = e->ePtr->pp.cPoint; /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

if (e->ePtr->pp.wpid2 == history\_edge)

{

getPolygonLine (e->ePtr->pp.lineCorner2, (Object) Indexing\_GetIndice (objects, e->ePtr->pp.pid2), &p1, &p2);

l = coord\_subCoord (p2, p1);

debugLine (p1, p2, (deviceIf\_Colour) deviceIf\_yellow ());

}

else

{

/\* hits corner, so we use the normal from the corner to the C of G of the polygon. \*/

l = coord\_subCoord (p, fixed->p.cOfG);

}

coord\_perpendiculars (l, &n, &n2); /\* n and n2 are normal vectors to the vector l. \*/

v = rotationalVelocity (moving->angularVelocity, coord\_initCoord (moving->vx, moving->vy), coord\_subCoord (p, moving->p.cOfG)); /\* n and n2 are normal vectors to the vector l. \*/

rap = coord\_subCoord (p, moving->p.cOfG);

rapn = coord\_perpendicular (rap);

I\_ = (sqr (coord\_dotProd (rapn, n)))/moving->inertia;

m = 1.0/moving->p.mass;

debugCircle (moving->p.cOfG, 0.002, (deviceIf\_Colour) deviceIf\_yellow ()); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* c of g for id1 \*/

debugCircle (fixed->p.cOfG, 0.002, (deviceIf\_Colour) deviceIf\_purple ()); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* c of g for id2 \*/

j = (-(1.0\*(coord\_dotProd (v, n))))/(((coord\_dotProd (n, n))\*m)+I\_); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* c of g for id2 \*/

updatePolygonVelocity (moving, -j, n, rapn);

deviceIf\_flipBuffer (); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

}

/\*

collidePolygonAgainstMovingPolygon - both, id1, and, id2, are moving.

\*/

static void collidePolygonAgainstMovingPolygon (eventQueue e, Object id1, Object id2)

{

coord\_Coord p1;

coord\_Coord p2;

coord\_Coord p;

coord\_Coord n;

coord\_Coord n2;

coord\_Coord v1;

coord\_Coord va;

coord\_Coord vb;

coord\_Coord vap;

coord\_Coord vbp;

coord\_Coord vab;

coord\_Coord rap;

coord\_Coord rbp;

coord\_Coord sum;

coord\_Coord vF1;

coord\_Coord vF2;

double ca;

double cb;

double denominator;

double vabDotN;

double modifiedVel;

double j1;

double j2;

double m;

if (PolygonDebugging)

{

libc\_printf ((char \*) "before processing collision\\n", 29);

displayEvent (e);

dumpObject (id1);

dumpObject (id2);

}

Assert (e->ePtr->etype == polygonPolygonEvent, 4389);

p = e->ePtr->pp.cPoint;

if (PolygonDebugging)

{

dumpDesc (e->ePtr);

deviceIf\_frameNote (); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

drawFrame (e); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

if ((e->ePtr->pp.wpid1 == history\_edge) && (e->ePtr->pp.wpid2 == history\_edge))

{

/\* avoid dangling else. \*/

if (PolygonDebugging)

{

libc\_printf ((char \*) "the edges of two polygon collide\\n", 34);

}

/\* choose the fixed line, if one exists. \*/

if (twoDsim\_isFixed (id1->id))

{

if (PolygonDebugging)

{

libc\_printf ((char \*) "using line %d:%d\\n", 18, e->ePtr->pp.pid1, e->ePtr->pp.lineCorner1);

}

getPolygonLine (e->ePtr->pp.lineCorner1, (Object) Indexing\_GetIndice (objects, e->ePtr->pp.pid1), &p1, &p2);

}

else

{

if (PolygonDebugging)

{

libc\_printf ((char \*) "using line %d:%d\\n", 18, e->ePtr->pp.pid2, e->ePtr->pp.lineCorner2);

}

getPolygonLine (e->ePtr->pp.lineCorner2, (Object) Indexing\_GetIndice (objects, e->ePtr->pp.pid2), &p1, &p2);

}

/\* p1 and p2 are the start end positions of the line \*/

v1 = coord\_subCoord (p2, p1); /\* v1 is the vector p1 -> p2 \*/

coord\_perpendiculars (v1, &n, &n2); /\* n and n2 are normal vectors to the vector v1 \*/

/\* n needs to point into id1. \*/

if (PolygonDebugging)

{

debugLine (p1, p2, (deviceIf\_Colour) deviceIf\_yellow ());

}

}

else if (e->ePtr->pp.wpid1 == history\_edge)

{

/\* avoid dangling else. \*/

/\* corner collision. \*/

if (PolygonDebugging)

{

libc\_printf ((char \*) "the edge of polygon collides with corner of polygon\\n", 53);

}

Assert (e->ePtr->pp.wpid2 == history\_corner, 4436);

getPolygonLine (e->ePtr->pp.lineCorner1, (Object) Indexing\_GetIndice (objects, e->ePtr->pp.pid1), &p1, &p2);

v1 = coord\_subCoord (p2, p1); /\* v1 is the vector p1 -> p2 \*/

coord\_perpendiculars (v1, &n, &n2); /\* n and n2 are normal vectors to the vector v1 \*/

/\* n needs to point into id1. \*/

if (PolygonDebugging)

{

debugLine (p1, p2, (deviceIf\_Colour) deviceIf\_purple ());

}

}

else if (e->ePtr->pp.wpid2 == history\_edge)

{

/\* avoid dangling else. \*/

if (PolygonDebugging)

{

libc\_printf ((char \*) "the edge of polygon collides with corner of polygon\\n", 53);

}

Assert (e->ePtr->pp.wpid1 == history\_corner, 4452);

getPolygonLine (e->ePtr->pp.lineCorner2, (Object) Indexing\_GetIndice (objects, e->ePtr->pp.pid2), &p1, &p2);

v1 = coord\_subCoord (p2, p1); /\* v1 is the vector p1 -> p2 \*/

coord\_perpendiculars (v1, &n, &n2); /\* n and n2 are normal vectors to the vector v1 \*/

/\* n needs to point into id1. \*/

if (PolygonDebugging)

{

debugLine (p1, p2, (deviceIf\_Colour) deviceIf\_white ());

}

}

else

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "the corners of two polygon collide\\n", 36);

/\* n needs and assignment. \*/

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

}

if (PolygonDebugging)

{

libc\_printf ((char \*) "line: ", 6);

dumpCoord (p1);

libc\_printf ((char \*) " -> ", 4);

dumpCoord (p2);

libc\_printf ((char \*) "\\n", 2);

libc\_printf ((char \*) "v1 vector along line: ", 22);

dumpCoord (v1);

libc\_printf ((char \*) " perpendicular: ", 17);

dumpCoord (n);

libc\_printf ((char \*) "\\n", 2);

debugCircle (id1->p.cOfG, 0.02, (deviceIf\_Colour) deviceIf\_yellow ()); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* c of g for id1 \*/

debugCircle (id2->p.cOfG, 0.02, (deviceIf\_Colour) deviceIf\_purple ()); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* c of g for id2 \*/

debugCircle (p, 0.02, (deviceIf\_Colour) deviceIf\_white ()); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* collision point \*/

}

/\* calculate relative velocity. \*/

rap = coord\_subCoord (p, id1->p.cOfG);

rbp = coord\_subCoord (p, id2->p.cOfG);

vap = coord\_addCoord (coord\_initCoord (id1->vx, id1->vy), coord\_scaleCoord (rap, id1->angularVelocity));

vbp = coord\_addCoord (coord\_initCoord (id2->vx, id2->vy), coord\_scaleCoord (rbp, id2->angularVelocity));

vab = coord\_subCoord (vap, vbp); /\* eq 1. C.Hecker. \*/

if (PolygonDebugging) /\* eq 1. C.Hecker. \*/

{

libc\_printf ((char \*) "v1 = ", 5);

dumpCoord (v1);

libc\_printf ((char \*) " rap = ", 7);

dumpCoord (rap);

libc\_printf ((char \*) " rbp = ", 7);

dumpCoord (rbp);

libc\_printf ((char \*) " vap = ", 7);

dumpCoord (vap);

libc\_printf ((char \*) " vbp = ", 7);

dumpCoord (vbp);

libc\_printf ((char \*) "\\n", 2);

libc\_printf ((char \*) "vab = ", 6);

dumpCoord (vab);

libc\_printf ((char \*) ", n = ", 6);

dumpCoord (n);

libc\_printf ((char \*) " dotProd (vab, n) = %g\\n", 24, coord\_dotProd (vab, n));

}

/\* eq 2. C.Hecker. \*/

if ((coord\_dotProd (vab, n)) < 0.0)

{

if (PolygonDebugging)

{

deviceIf\_flipBuffer (); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

libc\_printf ((char \*) "objects are moving apart, ignore.\\n", 35); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

}

/\* objects are moving apart, ignore. \*/

return;

}

/\* calculate impulse factor. Eq 6. C.Hecker. \*/

if (id1->fixed)

{

m = 0.0;

}

else

{

m = 1.0/id1->p.mass;

}

if (! id2->fixed)

{

m = m+(1.0/id2->p.mass);

}

denominator = (coord\_dotProd (n, n))\*m; /\* bottom of Eq 6. C.Hecker. \*/

if (PolygonDebugging) /\* bottom of Eq 6. C.Hecker. \*/

{

libc\_printf ((char \*) "n = ", 4);

dumpCoord (n);

libc\_printf ((char \*) "\\n", 2);

libc\_printf ((char \*) "m = %g, dotProd (n, n) = %g\\n", 29, m, coord\_dotProd (n, n));

libc\_printf ((char \*) "denominator = %g\\n", 18, denominator);

}

/\* calculate angular factors. \*/

if (id1->fixed)

{

ca = 0.0;

}

else

{

ca = (sqr (coord\_dotProd (coord\_perpendicular (rap), n)))/id1->inertia;

}

if (id2->fixed)

{

cb = 0.0;

}

else

{

cb = (sqr (coord\_dotProd (coord\_perpendicular (rbp), n)))/id2->inertia;

}

denominator = (denominator+ca)+cb;

/\* calculate total impulse of collision, j. \*/

vabDotN = coord\_dotProd (vab, n);

if (PolygonDebugging)

{

libc\_printf ((char \*) "vabDotN = %g\\n", 14, vabDotN);

}

modifiedVel = vabDotN/denominator;

j1 = -((1.0+Elasticity)\*modifiedVel);

j2 = (1.0+Elasticity)\*modifiedVel;

if (PolygonDebugging)

{

libc\_printf ((char \*) "j1 = %g, j2 = %g\\n", 18, j1, j2);

}

/\* update the velocities. \*/

if (! id1->fixed)

{

vF1 = coord\_addCoord (coord\_initCoord (id1->vx, id1->vy), coord\_scaleCoord (n, j1/id1->p.mass));

id1->vx = vF1.x;

id1->vy = vF1.y;

}

if (! id2->fixed)

{

vF2 = coord\_addCoord (coord\_initCoord (id2->vx, id2->vy), coord\_scaleCoord (n, j2/id2->p.mass));

id2->vx = vF2.x;

id2->vy = vF2.y;

}

/\* update the angular velocities. \*/

if (id1->fixed)

{

j2 = j2\*2.0;

}

if (id2->fixed)

{

j1 = j1\*2.0;

}

if (! id1->fixed)

{

id1->angularMomentum = id1->angularMomentum+(coord\_dotProd (rap, coord\_scaleCoord (n, j1)));

id1->angularVelocity = (1.0/id1->inertia)\*id1->angularMomentum;

}

if (! id2->fixed)

{

id2->angularMomentum = id2->angularMomentum+(coord\_dotProd (rbp, coord\_scaleCoord (n, j2)));

id2->angularVelocity = (1.0/id2->inertia)\*id2->angularMomentum;

}

if (PolygonDebugging)

{

libc\_printf ((char \*) "after processing collision\\n", 28);

displayEvent (e);

dumpObject (id1);

dumpObject (id2);

deviceIf\_flipBuffer (); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

dumpDesc (e->ePtr); /\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

if (! id2->fixed)

{

Assert (id2->vy > 0.0, 4610);

if (id2->vy > 0.0)

{

libc\_printf ((char \*) "SUCCESSFUL collide polygon\\n", 28);

}

else

{

debugDelay ((char \*) "FAILED collide polygon - terminating", 36);

}

}

if (! id1->fixed)

{

Assert (id1->vy > 0.0, 4620);

if (id1->vy > 0.0)

{

libc\_printf ((char \*) "SUCCESSFUL collide polygon\\n", 28);

}

else

{

debugDelay ((char \*) "FAILED collide polygon - terminating", 36);

}

}

}

}

/\*

physicsCollision - handle the physics of a collision between

the two objects defined in, e.

\*/

static void physicsCollision (eventQueue e)

{

Object id1;

Object id2;

switch (e->ePtr->etype)

{

case circlesEvent:

id1 = Indexing\_GetIndice (objects, e->ePtr->cc.cid1);

id2 = Indexing\_GetIndice (objects, e->ePtr->cc.cid2);

circleCollision (id1, id2);

break;

case circlePolygonEvent:

id1 = Indexing\_GetIndice (objects, e->ePtr->cp.cid);

id2 = Indexing\_GetIndice (objects, e->ePtr->cp.pid);

circlePolygonCollision (e, id1, id2);

break;

case polygonPolygonEvent:

id1 = Indexing\_GetIndice (objects, e->ePtr->pp.pid1);

id2 = Indexing\_GetIndice (objects, e->ePtr->pp.pid2);

polygonPolygonCollision (e, id1, id2);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

doCollision - called whenever a collision event is processed.

\*/

static void doCollision (eventQueue e)

{

updatePhysics (TRUE);

collisionOccurred (e->ePtr);

if (drawCollisionFrame)

{

if (Debugging)

{

libc\_printf ((char \*) "issuing collision draw frame\\n", 30);

}

deviceIf\_frameNote ();

drawFrame (e);

deviceIf\_flipBuffer ();

}

/\*

collectAll

\*/

physicsCollision (e);

/\*

printf ("near end of doCollision

");

printQueue ;

\*/

addNextObjectEvent ();

}

/\*

sqr -

\*/

static double sqr (double v)

{

return v\*v;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

cub -

\*/

static double cub (double v)

{

return (v\*v)\*v;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

quad -

\*/

static double quad (double v)

{

return ((v\*v)\*v)\*v;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

pent -

\*/

static double pent (double v)

{

return (quad (v))\*v;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

hex -

\*/

static double hex (double v)

{

return (cub (v))\*(cub (v));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

sept -

\*/

static double sept (double v)

{

return (quad (v))\*(cub (v));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

oct -

\*/

static double oct (double v)

{

return (quad (v))\*(quad (v));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getCircleValues - assumes, o, is a circle and retrieves:

center (x, y)

radius radius

velocity (vx, vy)

accel (ax, ay)

\*/

static void getCircleValues (Object o, double \*x, double \*y, double \*radius, double \*vx, double \*vy, double \*ax, double \*ay)

{

(\*x) = o->c.pos.x;

(\*y) = o->c.pos.y;

(\*radius) = o->c.r;

getObjectValues (o, vx, vy, ax, ay);

}

/\*

getObjectValues - fills in velocity and acceleration x, y, values.

\*/

static void getObjectValues (Object o, double \*vx, double \*vy, double \*ax, double \*ay)

{

coord\_Coord ac;

if (o->fixed || o->stationary)

{

(\*vx) = 0.0;

(\*vy) = 0.0;

(\*ax) = 0.0;

(\*ay) = 0.0;

}

else

{

ac = getAccelCoord (o);

(\*vx) = o->vx;

(\*vy) = o->vy;

(\*ax) = ac.x;

(\*ay) = ac.y;

}

}

/\*

getObjectOrbitingValues -

\*/

static void getObjectOrbitingValues (Object o, double \*r, double \*w, coord\_Coord \*cofg)

{

(\*r) = o->angleOrientation;

(\*w) = o->angularVelocity;

switch (o->object)

{

case polygonOb:

(\*cofg) = o->p.cOfG;

break;

case circleOb:

(\*cofg) = o->c.pos;

break;

default:

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

break;

}

}

/\*

maximaCircleCollisionOrbiting -

x1 y1 x2 y2

a, g, l, r is initial position of the point (not the c of g)

b, h, m, s is initial velocity

c, i, n, u is acceleration

e, k, p, v is angular velocity

f, q, k, w is the initial angular offset for the center of circle relative to the c of g.

The c of g is the center of the orbit.

d, o the distance of the point from the c of g.

\*/

static void maximaCircleCollisionOrbiting (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double i, double j, double k, double l, double m, double n, double o, double p, double q, double r, double s, double u, double v, double w, double x)

{

}

/\*

earlierCircleCollisionOrbiting -

t is the time of this collision (if any)

tc is the time of the next collision.

c1p is the initial position of the center of circle 1.

This may not be the c of g of this circle if it is in orbit.

c1radius is the radius of the circle.

c1r rotational offset of point 1.

c1cofg center of gravity of point 1.

c1w rotational angular velocity of point 1.

c1a acceleration of point 1.

c2p is the initial position of the center of circle 2.

This may not be the c of g of this circle if it is in orbit.

c2radius is the radius of the circle.

c2r rotational offset of point 2.

c2cofg center of gravity of point 2.

c2w rotational angular velocity of point 2.

c2a acceleration of point 2.

\*/

static unsigned int earlierCircleCollisionOrbiting (double \*t, double \*tc, coord\_Coord c1p, double c1radius, double c1r, double c1w, coord\_Coord c1cofg, coord\_Coord c1v, coord\_Coord c1a, coord\_Coord c2p, double c2radius, double c2r, double c2w, coord\_Coord c2cofg, coord\_Coord c2v, coord\_Coord c2a)

{

typedef struct \_T13\_a \_T13;

struct \_T13\_a { double array[8+1]; };

double d1;

double d2;

double A;

double B;

double C;

double D;

double E;

double F;

double G;

double H;

double I\_;

double T;

\_T13 array;

d1 = coord\_lengthCoord (coord\_subCoord (c1p, c1cofg));

d2 = coord\_lengthCoord (coord\_subCoord (c2p, c2cofg));

/\* describe line 2 by its Y coordinates. \*/

maximaCircleCollisionOrbiting ((double \*) &array.array[0], 8, c1p.x, c1v.x, c1a.x, d1, c1w, c1r, c2p.x, c2v.x, c2a.x, c2w, c2r, c1p.y, c1v.y, c1a.y, d2, c1w, c1r, c2p.y, c2v.y, c2a.y, c2w, c2r, c1radius+c2radius);

A = array.array[8];

B = array.array[7];

C = array.array[6];

D = array.array[5];

E = array.array[4];

F = array.array[3];

G = array.array[2];

H = array.array[1];

I\_ = array.array[0];

/\* now solve for values of t which satisfy:

At^8 + Bt^7 + Ct^6 + Dt^5 + Et^4 + Ft^3 + Gt^2 + Ht + I = 0 \*/

if (roots\_findOctic (A, B, C, D, E, F, G, H, I\_, t))

{

T = ((((((((A\*(oct ((\*t))))+(B\*(sept ((\*t)))))+(C\*(hex ((\*t)))))+(D\*(pent ((\*t)))))+(E\*(quad ((\*t)))))+(F\*(cub ((\*t)))))+(G\*(sqr ((\*t)))))+(H\*(\*t)))+I\_;

if (Debugging)

{

libc\_printf ((char \*) "%gt^8 + %gt^7 +%gt^6 + %gt^5 + %gt^4 + %gt^3 + %gt^2 + %gt + %g = %g (t=%g)\\n", 80, A, B, C, D, E, F, G, H, I\_, T, (\*t));

}

/\* remember tc is -1.0 initially, to force it to be set once. \*/

if ((((\*tc) < 0.0) || ((\*t) < (\*tc))) && (! (roots\_nearZero ((\*t)))))

{

return TRUE;

}

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

maximaCircleCollision -

\*/

static void maximaCircleCollision (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p)

{

}

static unsigned int earlierCircleCollision (eventDesc edesc, unsigned int id1, unsigned int id2, double \*t, double bestTimeOfCollision, coord\_Coord \*cp, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p)

{

double T;

double S;

double W;

double gap;

/\*

IF (bestTimeOfCollision >= 0.0) AND (t >= 0.0)

THEN

gap := 0.5 ;

T := 1.0 / framesPerSecond ;

S := sqr ((d - c) \* T + 0.5 \* (e - f) \* sqr (T)) +

sqr ((k - l) \* T + 0.5 \* (m - n) \* sqr (T)) ; estimated distance travelled in next time frame.

W := sqr (a - b) + sqr (g - h) - sqr (o + p) ; current distance between the two circles.

IF S + gap < W

THEN

INC (noOfCulledCollisions) ;

IF noOfCulledCollisions MOD 100 = 0

THEN

printf ("have culled %d collisions

", noOfCulledCollisions)

END ;

RETURN FALSE

END

END ;

\*/

return doEarlierCircleCollision (edesc, id1, id2, t, bestTimeOfCollision, cp, a, b, c, d, e, f, g, h, k, l, m, n, o, p);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doEarlierCircleCollision - let the following abreviations be assigned.

Note i is one circle, j is another circle.

v is velocity, a, acceleration, x, y axis.

r is radius.

Single letter variables are used since wxmaxima

only operates with these. Thus the output from wxmaxima

can be cut and pasted into the program.

a = xi

b = xj

c = vxi

d = vxj

e = aix

f = ajx

g = yi

h = yj

k = vyi

l = vyj

m = aiy

n = ajy

o = ri

p = rj

t is the time of this collision (if any)

bestTimeOfCollision is earlier known collision so far.

\*/

static unsigned int doEarlierCircleCollision (eventDesc edesc, unsigned int id1, unsigned int id2, double \*t, double bestTimeOfCollision, coord\_Coord \*cp, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p)

{

typedef struct \_T14\_a \_T14;

struct \_T14\_a { double array[4+1]; };

double A;

double B;

double C;

double D;

double E;

double T;

\_T14 array;

coord\_Coord c1;

coord\_Coord c2;

coord\_Coord cp1;

coord\_Coord cp2;

coord\_Coord v12;

coord\_Coord r12;

/\* thanks to wxmaxima (expand ; factor ; ratsimp) \*/

A = (((((sqr (n))-((2.0\*m)\*n))+(sqr (m)))+(sqr (f)))-((2.0\*e)\*f))+(sqr (e));

B = (((((4.0\*l)-(4.0\*k))\*n)+(((4.0\*k)-(4.0\*l))\*m))+(((4.0\*d)-(4.0\*c))\*f))+(((4.0\*c)-(4.0\*d))\*e);

C = (((((((((((4.0\*h)-(4.0\*g))\*n)+(((4.0\*g)-(4.0\*h))\*m))+(4.0\*(sqr (l))))-((8.0\*k)\*l))+(4.0\*(sqr (k))))+(((4.0\*b)-(4.0\*a))\*f))+(((4.0\*a)-(4.0\*b))\*e))+(4.0\*(sqr (d))))-((8.0\*c)\*d))+(4.0\*(sqr (c)));

D = (((((8.0\*h)-(8.0\*g))\*l)+(((8.0\*g)-(8.0\*h))\*k))+(((8.0\*b)-(8.0\*a))\*d))+(((8.0\*a)-(8.0\*b))\*c);

E = ((((((4.0\*(sqr (h)))-((8.0\*g)\*h))+(4.0\*(sqr (g))))+(4.0\*(sqr (b))))-((8.0\*a)\*b))+(4.0\*(sqr (a))))-(sqr (2.0\*(p+o)));

/\*

maximaCircleCollision (array,

a, b, c, d, e, f, g, h, k, l, m, n, o, p) ;

\*/

manualCircleCollision ((double \*) &array.array[0], 4, a, b, c, d, e, f, g, h, k, l, m, n, o, p);

AssertRDebug (array.array[4], A, (char \*) "A", 1);

AssertRDebug (array.array[3], B, (char \*) "B", 1);

AssertRDebug (array.array[2], C, (char \*) "C", 1);

AssertRDebug (array.array[1], D, (char \*) "D", 1);

AssertRDebug (array.array[0], E, (char \*) "E", 1);

/\* now solve for values of t which satisfy At^4 + Bt^3 + Ct^2 + Dt^1 + Et^0 = 0 \*/

if (roots\_findQuartic (A, B, C, D, E, t)) /\* this function will alter, t. \*/

{

T = ((((A\*((sqr ((\*t)))\*(sqr ((\*t)))))+(B\*((sqr ((\*t)))\*(\*t))))+(C\*(sqr ((\*t)))))+(D\*(\*t)))+E;

if (Debugging)

{

libc\_printf ((char \*) "%gt^4 + %gt^3 +%gt^2 + %gt + %g = %g (t=%g)\\n", 48, A, B, C, D, E, T, (\*t));

libc\_printf ((char \*) "found collision at %g\\n", 23, (\*t));

}

Assert ((\*t) >= 0.0, 5068);

/\* remember edesc = NIL if bestTimeOfCollision is unassigned. \*/

if ((edesc == NULL) || ((\*t) < bestTimeOfCollision))

{

c1 = newPositionCoord (coord\_initCoord (a, g), coord\_initCoord (c, k), coord\_initCoord (e, m), (\*t));

c2 = newPositionCoord (coord\_initCoord (b, h), coord\_initCoord (d, l), coord\_initCoord (f, n), (\*t));

v12 = coord\_subCoord (c1, c2);

Assert (roots\_nearCoord (c1, coord\_addCoord (c2, v12)), 5079);

cp2 = coord\_addCoord (c2, coord\_scaleCoord (v12, o/(o+p)));

cp1 = coord\_subCoord (c1, coord\_scaleCoord (v12, p/(o+p)));

(\*cp) = cp2;

if (roots\_nearSame (coord\_lengthCoord (v12), o+p))

{

/\* avoid dangling else. \*/

/\*

printf ("

c1 = %g, %g

", c1.x, c1.y) ;

printf ("c2 = %g, %g

", c2.x, c2.y) ;

printf ("o = %g, p = %g

", o, p) ;

printf ("cp2 = %g, %g

", cp2.x, cp2.y) ;

printf ("v12 = c1 - c2 = %g, %g

", v12.x, v12.y) ;

r12 := scaleCoord (v12, o/(o+p)) ;

printf ("r12 = v12 scaled by %g = %g, %g

", o/(o+p), r12.x, r12.y) ;

printf ("cp1 = c2 + r12 = %g, %g

", cp1.x, cp1.y) ;

\*/

Assert (roots\_nearCoord (cp1, cp2), 5096);

/\* found a value of t which is better than bestTimeOfCollision, but it might be a duplicate collision. \*/

if (! (history\_isDuplicateC (currentTime, (\*t), id1, id2, (history\_whereHit) history\_edge, (history\_whereHit) history\_edge, (\*cp))))

{

/\* ok, this has not been seen before. \*/

return TRUE;

}

}

else

{

if (trace)

{

libc\_printf ((char \*) "false the collisions points do not touch ignoring = %g, %g\\n", 60, (\*cp).x, (\*cp).y);

}

}

}

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

findCollisionCircles -

using:

S = UT + (AT^2)/2

compute xin and yin which are the new (x,y) positions of object i at time, t.

compute xjn and yjn which are the new (x,y) positions of object j at time, t.

now compute difference between objects and if they are ri+rj (radius of circle, i, and, j)

apart then we have a collision at time, t.

xin = xi + vxi \* t + (aix \* t^2) / 2.0

yin = yi + vyi \* t + (aiy \* t^2) / 2.0

xjn = xj + vxj \* t + (ajx \* t^2) / 2.0

yjn = yj + vyj \* t + (ajy \* t^2) / 2.0

ri + rj == sqrt(abs(xin-xjn)^2 + abs(yin-yjn)^2) for values of t

ri + rj == sqrt(((xi + vxi \* t + aix \* t^2 / 2.0) - (xj + vxj \* t + ajx \* t^2 / 2.0))^2 +

((yi + vyi \* t + aiy \* t^2 / 2.0) - (yj + vyj \* t + ajy \* t^2 / 2.0))^2)

let:

a = xi

b = xj

c = vxi

d = vxj

e = aix

f = ajx

g = yi

h = yj

k = vyi

l = vyj

m = aiy

n = ajy

o = ri

p = rj

t = t

o + p == sqrt(((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2)

o + p == sqrt(((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2)

0 == ((a + c \* t + e \* t^2 / 2.0) - (b + d \* t + f \* t^2 / 2.0))^2 +

((g + k \* t + m \* t^2 / 2.0) - (h + l \* t + n \* t^2 / 2.0))^2 -

(o + p)^2

now using wxmaxima

expand ; factor ; ratsimp

p+o == (sqrt((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))/2

2\*(p+o) == (sqrt((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))

(2\*(p+o))^2 == ((n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2))

0 == (n^2-2\*m\*n+m^2+f^2-2\*e\*f+e^2)\*t^4+

((4\*l-4\*k)\*n+(4\*k-4\*l)\*m+(4\*d-4\*c)\*f+(4\*c-4\*d)\*e)\*t^3+

((4\*h-4\*g)\*n+(4\*g-4\*h)\*m+4\*l^2-8\*k\*l+4\*k^2+(4\*b-4\*a)\*f+(4\*a-4\*b)\*e+4\*d^2-8\*c\*d+4\*c^2)\*t^2+

((8\*h-8\*g)\*l+(8\*g-8\*h)\*k+(8\*b-8\*a)\*d+(8\*a-8\*b)\*c)\*t+

4\*h^2-8\*g\*h+4\*g^2+4\*b^2-8\*a\*b+4\*a^2)-

((2\*(p+o))^2)

solve polynomial:

A := sqr(n)-2.0\*m\*n+sqr(m)+sqr(f)-2.0\*e\*f+sqr(e) ;

B := (4.0\*l-4.0\*k)\*n+(4.0\*k-4.0\*l)\*m+(4.0\*d-4.0\*c)\*f+(4.0\*c-4.0\*d)\*e ;

C := (4.0\*h-4.0\*g)\*n+(4.0\*g-4.0\*h)\*m+4.0\*sqr(l)-8.0\*k\*l+4.0\*sqr(k)+(4.0\*b-4.0\*a)\*f+(4.0\*a-4.0\*b)\*e+4.0\*sqr(d)-8.0\*c\*d+4.0\*sqr(c) ;

D := (8.0\*h-8.0\*g)\*l+(8.0\*g-8.0\*h)\*k+(8.0\*b-8.0\*a)\*d+(8.0\*a-8.0\*b)\*c ;

E := 4.0\*sqr(h)-8.0\*g\*h+4.0\*sqr(g)+4.0\*sqr(b)-8.0\*a\*b+4.0\*sqr(a)-sqr(2.0\*(p+o)) ;

\*/

static void findCollisionCircles (Object iptr, Object jptr, eventDesc \*edesc, double \*tc)

{

double a;

double b;

double c;

double d;

double e;

double f;

double g;

double h;

double k;

double l;

double m;

double n;

double o;

double p;

double t;

unsigned int i;

unsigned int j;

double T;

coord\_Coord cp;

/\*

a xi

g yi

o ri

c vxi

k vyi

e axi

m ayi

\*/

getCircleValues (iptr, &a, &g, &o, &c, &k, &e, &m);

/\*

b xj

h yj

p rj

d vxj

l vyj

f ajx

n ajy

\*/

getCircleValues (jptr, &b, &h, &p, &d, &l, &f, &n);

if (earlierCircleCollision ((\*edesc), iptr->id, jptr->id, &t, (\*tc), &cp, a, b, c, d, e, f, g, h, k, l, m, n, o, p))

{

(\*tc) = t;

(\*edesc) = makeCirclesDesc (edesc, iptr->id, jptr->id, cp);

}

}

/\*

stop -

\*/

static void stop (void)

{

}

/\*

makeCirclesPolygonDesc - returns an eventDesc describing the collision between a circle and a polygon.

notice wpid1 is not used, this is because it is called indirectly

and there are other make...Desc functions which have exactly the

same parameter data types.

\*/

static eventDesc makeCirclesPolygonDesc (eventDesc edesc, unsigned int cid, unsigned int pid, unsigned int lineNo, unsigned int pointNo, history\_whereHit wpid1, history\_whereHit wpid2, coord\_Coord collisionPoint)

{

/\* circle must always be treated as corner. \*/

if (edesc == NULL)

{

edesc = newDesc ();

}

edesc->etype = circlePolygonEvent;

edesc->cp.pid = pid;

edesc->cp.cid = cid;

edesc->cp.cPoint = collisionPoint;

edesc->cp.wpid = wpid2;

edesc->cp.lineNo = lineNo;

edesc->cp.pointNo = pointNo;

return edesc;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

checkIfPointHits - return TRUE if t0 is either the first hit found or

is sooner than, tc. It determines a hit by working out

the final position of partical:

x = s + ut + 1/2a t^2

if x>=0.0 and x <= length then it hits.

\*/

static unsigned int checkIfPointHits (double timeOfPrevCollision, double t, double length, double s, double u, double a)

{

double x;

/\* if t is later than timeOfPrevCollision, then we don't care as we already have found an earlier hit. \*/

if (trace)

{

libc\_printf ((char \*) "current best collision time is %g the new collision exists at time %g\\n", 71, timeOfPrevCollision, t);

}

if ((timeOfPrevCollision == -1.0) || (t < timeOfPrevCollision))

{

/\* avoid dangling else. \*/

/\* at time, t, what is the value of x ? \*/

x = newPositionScalar (s, u, a, t);

if (trace)

{

libc\_printf ((char \*) "line 0.0 .. %g and point at %g ", 32, length, x);

}

/\* if x lies between 0 .. length then it hits! \*/

if ((x >= 0.0) && (x <= length))

{

/\* new earlier collision time found \*/

if (trace)

{

libc\_printf ((char \*) "will hit line\\n", 15);

}

return TRUE;

}

if (trace)

{

libc\_printf ((char \*) "misses line\\n", 13);

}

}

else if (trace)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "the new time %g should be ignored as the best is %g\\n", 53, t, timeOfPrevCollision);

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newPositionScalar - calculates the new position of a scalar in the future.

\*/

static double newPositionScalar (double s, double u, double a, double t)

{

return (s+(u\*t))+((a\*(t\*t))/2.0);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newPositionRotationSinScalar - works out the new Y position for a point whose:

current cofg Y position is: c

initial Y velocity is : u

Y acceleration is : a

angular velocity : w

orientation : o

polar coord position rel

to cofg is : p

\*/

static double newPositionRotationSinScalar (double c, double u, double a, double t, double w, double o, polar\_Polar p)

{

double O;

if (Debugging)

{

libc\_printf ((char \*) "c = %g, u = %g, a = %g, t = %g\\n", 32, c, u, a, t);

}

O = newPositionScalar (c, u, a, t);

if (Debugging)

{

libc\_printf ((char \*) "O = %g, p.r = %g, p.w = %g, sin (w\*t + o + p.w) = %g\\n", 54, O, p.r, p.w, libm\_sin (((w\*t)+o)+p.w));

}

return O+(p.r\*(libm\_sin (((w\*t)+o)+p.w)));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newPositionRotationCosScalar - works out the new X position for a point whose:

current cofg X position is: c

initial X velocity is : u

X acceleration is : a

angular velocity : w

orientation : o

polar coord position rel

to cofg is : p

\*/

static double newPositionRotationCosScalar (double c, double u, double a, double t, double w, double o, polar\_Polar p)

{

double O;

O = newPositionScalar (c, u, a, t);

return O+(p.r\*(libm\_cos (((w\*t)+o)+p.w)));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newPositionCoord - calculates the new position of point in the future.

\*/

static coord\_Coord newPositionCoord (coord\_Coord c, coord\_Coord u, coord\_Coord a, double t)

{

return coord\_initCoord (newPositionScalar (c.x, u.x, a.x, t), newPositionScalar (c.y, u.y, a.y, t));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newPositionRotationCoord - calculates the new position of point, c+v, in the future.

Given angular velocity : w

orientation : o

time : t

initial vel : u

accel : a

c of g : c

polar coord of the point : p

\*/

static coord\_Coord newPositionRotationCoord (coord\_Coord c, coord\_Coord u, coord\_Coord a, double t, double w, double o, polar\_Polar p)

{

if (Debugging)

{

libc\_printf ((char \*) "t = %g, w = %g, o = %g\\n", 24, t, w, o);

}

return coord\_initCoord (newPositionRotationCosScalar (c.x, u.x, a.x, t, w, o, p), newPositionRotationSinScalar (c.y, u.y, a.y, t, w, o, p));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

hLine - debugging procedure to display a line on a half scale axis.

\*/

static void hLine (coord\_Coord p1, coord\_Coord p2, deviceIf\_Colour c)

{

typedef struct \_T15\_a \_T15;

struct \_T15\_a { Points\_Point array[1+1]; };

\_T15 p;

p1 = coord\_scaleCoord (p1, 0.5);

p2 = coord\_scaleCoord (p2, 0.5);

p1 = coord\_addCoord (p1, coord\_initCoord (0.5, 0.5));

p2 = coord\_addCoord (p2, coord\_initCoord (0.5, 0.5));

p.array[0] = c2p (p1);

p.array[1] = c2p (p2);

deviceIf\_glyphPolygon (2, (Points\_Point \*) &p.array[0], 1, FALSE, Fractions\_initFract (0, 1, 16), c);

}

/\*

hPoint - debugging procedure to display a line on a half scale axis.

\*/

static void hPoint (coord\_Coord p, deviceIf\_Colour c)

{

p = coord\_scaleCoord (p, 0.5);

p = coord\_addCoord (p, coord\_initCoord (0.5, 0.5));

deviceIf\_glyphCircle (c2p (p), TRUE, Fractions\_zero (), Fractions\_putReal (0.05), c);

}

/\*

hCircle - debugging procedure to display a circle on a half scale axis.

\*/

static void hCircle (coord\_Coord p, double r, deviceIf\_Colour c)

{

p = coord\_scaleCoord (p, 0.5);

p = coord\_addCoord (p, coord\_initCoord (0.5, 0.5));

deviceIf\_glyphCircle (c2p (p), TRUE, Fractions\_zero (), Fractions\_putReal (r), c);

}

/\*

hVec - display a normalised vector on a half scale axis

\*/

static void hVec (coord\_Coord p, deviceIf\_Colour c)

{

p = coord\_normaliseCoord (p);

hLine (coord\_initCoord (0.0, 0.0), coord\_initCoord (p.x, 0.0), c);

hLine (coord\_initCoord (0.0, 0.0), coord\_initCoord (0.0, p.y), c);

}

/\*

hFlush - flip the debugging buffer.

\*/

static void hFlush (void)

{

deviceIf\_frameNote ();

drawBoarder ((deviceIf\_Colour) deviceIf\_black ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

/\*

checkPointCollision -

\*/

static unsigned int checkPointCollision (double \*timeOfPrevCollision, double t, double length, double cx, double rvx, double rax, coord\_Coord c, coord\_Coord cvel, coord\_Coord caccel, coord\_Coord \*collisionPoint, unsigned int id1, unsigned int id2)

{

if (trace)

{

libc\_printf ((char \*) "entering checkPointCollision with the current best time %g\\n", 60, (\*timeOfPrevCollision));

}

if (checkIfPointHits ((\*timeOfPrevCollision), t, length, cx, rvx, rax))

{

if (trace)

{

libc\_printf ((char \*) "it crosses the region of interest (current best time %g)\\n", 58, (\*timeOfPrevCollision));

}

/\* a hit, find where \*/

(\*collisionPoint) = newPositionCoord (c, cvel, caccel, t);

/\* return TRUE providing that we do not already know about it \*/

if (history\_isDuplicateC (currentTime, t, id1, id2, (history\_whereHit) history\_edge, (history\_whereHit) history\_edge, (\*collisionPoint)))

{

if (trace)

{

libc\_printf ((char \*) "but it is a duplicate (best time still %g)\\n", 44, (\*timeOfPrevCollision));

}

return FALSE;

}

else

{

if (trace)

{

libc\_printf ((char \*) "point hits line, new best time of collision (%g)\\n", 50, t);

}

(\*timeOfPrevCollision) = t;

return TRUE;

}

}

else

{

if (trace)

{

libc\_printf ((char \*) "the point missed the line (best time is still %g)\\n", 51, (\*timeOfPrevCollision));

}

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

earlierPointLineCollision - returns TRUE if we can find a collision between a point, c,

travelling at cvel with acceleration, caccel and a line

p1, p2, travelling at velocity, lvel, and acceleration laccel.

If a collision is found then the collisionPoint is also

calculated.

\*/

static unsigned int earlierPointLineCollision (double \*timeOfCollision, coord\_Coord c, coord\_Coord cvel, coord\_Coord caccel, coord\_Coord p1, coord\_Coord p2, coord\_Coord lvel, coord\_Coord laccel, coord\_Coord \*collisionPoint, unsigned int id1, unsigned int id2)

{

coord\_Coord p3;

coord\_Coord c0;

coord\_Coord c1;

coord\_Coord rvel;

coord\_Coord raccel;

double x;

double t0;

double t1;

double hypot;

double theta;

if (trace)

{

libc\_printf ((char \*) "earlierPointLineCollision entered and best time is %g\\n", 55, (\*timeOfCollision));

}

/\* we pretend that the line is stationary, by computing the relative velocity and acceleration \*/

rvel = coord\_subCoord (cvel, lvel);

raccel = coord\_subCoord (caccel, laccel);

if (trace)

{

libc\_printf ((char \*) "relative vel (%g, %g), accel (%g, %g)\\n", 41, rvel.x, rvel.y, raccel.x, raccel.y);

}

/\* now translate p1 onto the origin \*/

p3 = coord\_subCoord (p2, p1);

hypot = coord\_lengthCoord (p3);

/\* now find theta the angle of the vector, p3 \*/

theta = libm\_asin (p3.y/hypot);

if (trace)

{

libc\_printf ((char \*) "rotating line by %g degrees (length of line is %g)\\n", 53, (180.0\*theta)/3.14159, hypot);

}

c0 = coord\_subCoord (c, p1); /\* translate c by the same as the line \*/

c1 = coord\_rotateCoord (c0, -theta); /\* and rotate point, c0. \*/

rvel = coord\_rotateCoord (rvel, -theta); /\* and relative velocity \*/

raccel = coord\_rotateCoord (raccel, -theta); /\* and relative acceleration \*/

raccel = checkZeroCoord (raccel); /\* and relative acceleration \*/

rvel = checkZeroCoord (rvel);

if (trace)

{

libc\_printf ((char \*) "after rotation we have relative vel (%g, %g), accel (%g, %g)\\n", 64, rvel.x, rvel.y, raccel.x, raccel.y);

}

if (FALSE)

{

hLine (coord\_initCoord (0.0, 0.0), coord\_initCoord (hypot, 0.0), (deviceIf\_Colour) deviceIf\_purple ());

hPoint (c1, (deviceIf\_Colour) deviceIf\_purple ());

hFlush ();

}

/\*

now solve for, t, when y=0, use S = UT + 1/2 AT^2

at y = 0 we have:

0 = rvel.y \* t + 1/2 \* raccel.y \* t^2

Using quadratic:

at^2 + bt + c = 0

\*/

if (roots\_findQuadratic (raccel.y/2.0, rvel.y, c1.y, &t0, &t1))

{

if (trace)

{

libc\_printf ((char \*) "earlierPointLineCollision after findQuadratic and best time is %g\\n", 67, (\*timeOfCollision));

}

if ((t0 < 0.0) && (t1 < 0.0))

{

if (trace)

{

libc\_printf ((char \*) "the point never crosses the line in the future\\n", 48);

}

/\* get out of here quick - no point of predicting collisions in the past :-) \*/

return FALSE;

}

else

{

if (t0 == t1)

{

/\* avoid dangling else. \*/

if (trace)

{

libc\_printf ((char \*) "the point crosses the line once\\n", 33);

}

/\* only one root \*/

if (checkPointCollision (timeOfCollision, t0, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

return TRUE;

}

}

else

{

/\* two roots, ignore a negative root \*/

if (t0 < 0.0)

{

/\* avoid dangling else. \*/

if (trace)

{

libc\_printf ((char \*) "the point crosses the line once in the future and once in the past\\n", 68);

libc\_printf ((char \*) "only examining root %g, remember best is %g\\n", 45, t1, (\*timeOfCollision));

}

/\* test only positive root, t1 \*/

if (checkPointCollision (timeOfCollision, t1, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

return TRUE;

}

}

else if (t1 < 0.0)

{

/\* avoid dangling else. \*/

if (trace)

{

libc\_printf ((char \*) "the point crosses the line once in the future and once in the past\\n", 68);

}

/\* test only positive root, t0 \*/

if (checkPointCollision (timeOfCollision, t0, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

return TRUE;

}

}

else

{

/\* avoid dangling else. \*/

if (trace)

{

libc\_printf ((char \*) "the point crosses the line twice in the future\\n", 48);

}

/\* ok two positive roots, test smallest (earlist first and then bail out if it hits) \*/

if (t0 < t1)

{

/\* avoid dangling else. \*/

if (checkPointCollision (timeOfCollision, t0, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

if (trace)

{

libc\_printf ((char \*) "the point crosses the line first time in correct place\\n", 56);

}

return TRUE;

}

if (checkPointCollision (timeOfCollision, t1, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

if (trace)

{

libc\_printf ((char \*) "the point crosses the line, first time misses and second time hits\\n", 68);

}

return TRUE;

}

}

else

{

if (checkPointCollision (timeOfCollision, t1, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

if (trace)

{

libc\_printf ((char \*) "the point crosses the line first time in correct place\\n", 56);

}

return TRUE;

}

if (checkPointCollision (timeOfCollision, t0, hypot, c1.x, rvel.x, raccel.x, c, cvel, caccel, collisionPoint, id1, id2))

{

if (trace)

{

libc\_printf ((char \*) "the point crosses the line, first time misses and second time hits\\n", 68);

}

return TRUE;

}

}

}

}

}

}

if (trace)

{

libc\_printf ((char \*) "this point and line should be discarded\\n", 41);

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

sortLine - orders points, p1 and, p2, according to their x value.

\*/

static void sortLine (coord\_Coord \*p1, coord\_Coord \*p2)

{

coord\_Coord t;

if ((\*p1).x > (\*p2).x)

{

t = (\*p1);

(\*p1) = (\*p2);

(\*p2) = t;

}

else if (((\*p1).x == (\*p2).x) && ((\*p1).y > (\*p2).y))

{

/\* avoid dangling else. \*/

t = (\*p1);

(\*p1) = (\*p2);

(\*p2) = t;

}

}

/\*

findEarlierCircleEdgeCollision - return TRUE if an earlier time, t, is found than tc for when circle

hits a line. The circle is defined by a, center, radius and has

acceleration, accelCircle, and velocity, velCircle.

The line is between p1 and p2 and has velocity, velLine, and

acceleration, accelLine.

\*/

static void findEarlierCircleEdgeCollision (double \*timeOfCollision, unsigned int cid, unsigned int pid, unsigned int lineP, unsigned int lineC, eventDesc \*edesc, coord\_Coord center, double radius, coord\_Coord velCircle, coord\_Coord accelCircle, coord\_Coord p1, coord\_Coord p2, coord\_Coord velLine, coord\_Coord accelLine, descP createDesc)

{

coord\_Coord v1;

coord\_Coord n1;

coord\_Coord d1;

coord\_Coord d2;

coord\_Coord p3;

coord\_Coord p4;

coord\_Coord p5;

coord\_Coord p6;

coord\_Coord collisonPoint;

sortLine (&p1, &p2);

/\* create the vector p1 -> p2. \*/

v1 = coord\_subCoord (p2, p1);

/\* compute the normal for v1, normalise it, and multiply by radius. \*/

coord\_perpendiculars (v1, &d1, &d2);

d1 = coord\_scaleCoord (coord\_normaliseCoord (d1), radius);

d2 = coord\_scaleCoord (coord\_normaliseCoord (d2), radius);

/\* now add d1, d2 to p1 to obtain p3, p4. \*/

p3 = coord\_addCoord (p1, d1);

p4 = coord\_addCoord (p1, d2);

/\* now add d1 and d2 to p2 to get p5 and p6. \*/

p5 = coord\_addCoord (p2, d1);

p6 = coord\_addCoord (p2, d2);

/\* ok, now we only need to find when line between p3, p5 hits the centre of the circle. \*/

if (earlierPointLineCollision (timeOfCollision, center, velCircle, accelCircle, p3, p5, velLine, accelLine, &collisonPoint, cid, pid))

{

/\* circle hits line, p1, in tc seconds. \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits line (%g, %g) (%g, %g) in %g\\n", 42, p1.x, p1.y, p2.x, p2.y, (\*timeOfCollision));

}

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_edge, collisonPoint);

if (drawPrediction)

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (center, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugLine (p3, p5, (deviceIf\_Colour) deviceIf\_yellow ());

debugCircle (collisonPoint, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

/\* ok, now we only need to find when line between p4, p6 hits the centre of the circle. \*/

if (earlierPointLineCollision (timeOfCollision, center, velCircle, accelCircle, p4, p6, velLine, accelLine, &collisonPoint, cid, pid))

{

/\* circle hits line, p1, in tc seconds \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits line (%g, %g) (%g, %g) in %g\\n", 42, p1.x, p1.y, p2.x, p2.y, (\*timeOfCollision));

}

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_edge, collisonPoint);

if (drawPrediction)

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (center, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugLine (p4, p6, (deviceIf\_Colour) deviceIf\_yellow ());

debugCircle (collisonPoint, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

}

/\*

getPolygonCoord -

\*/

static coord\_Coord getPolygonCoord (Object pPtr, unsigned int pointno)

{

return checkZeroCoord (coord\_addCoord (pPtr->p.cOfG, polar\_polarToCoord (polar\_rotatePolar ((polar\_Polar) pPtr->p.points.array[pointno], pPtr->angleOrientation))));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getPolygonLine - assigns, c1, c2, with the, line, coordinates of polygon, pPtr.

\*/

static void getPolygonLine (unsigned int line, Object pPtr, coord\_Coord \*c1, coord\_Coord \*c2)

{

(\*c1) = getPolygonCoord (pPtr, line-1);

if (line == pPtr->p.nPoints)

{

(\*c2) = getPolygonCoord (pPtr, 0);

}

else

{

(\*c2) = getPolygonCoord (pPtr, line);

}

}

/\*

findCollisionCircleLine - find the time (if any) between line number, lineP, in polygon, pPtr,

and the circle, cPtr. cPtr can also be a polygon in which case lineC

is the particular line under question. Line on line collision is broken

down into circle line calls which allows for code reuse.

Neither cPtr or pPtr are orbiting.

\*/

static void findCollisionCircleLine (Object cPtr, Object pPtr, unsigned int lineP, unsigned int lineC, coord\_Coord center, double radius, eventDesc \*edesc, double \*timeOfCollision, descP createDesc)

{

coord\_Coord velCircle;

coord\_Coord accelCircle;

coord\_Coord velLine;

coord\_Coord accelLine;

coord\_Coord p1;

coord\_Coord p2;

coord\_Coord cp;

double cx;

double cy;

double r;

double cvx;

double cvy;

double cax;

double cay;

double pvx;

double pvy;

double pax;

double pay;

double t;

unsigned int cid;

unsigned int pid;

cid = cPtr->id;

pid = pPtr->id;

getPolygonLine (lineP, pPtr, &p1, &p2);

/\* we perform 4 checks.

(i) and (ii) pretend the circle has radius 0.0 and see if it hits two new circles at

point, p1, and, p2 with the original radius.

(iii) and (iv) now draw two lines between the edge of the two new circles and see if the

center of the original circle intersects with either line.

the smallest positive time is the time of the next collision.

\*/

getObjectValues (cPtr, &cvx, &cvy, &cax, &cay);

getObjectValues (pPtr, &pvx, &pvy, &pax, &pay);

/\* i \*/

if (earlierCircleCollision ((\*edesc), cid, pid, &t, (\*timeOfCollision), &cp, p1.x, center.x, pvx, cvx, pax, cax, p1.y, center.y, pvy, cvy, pay, cay, radius, 0.0))

{

/\* circle hits corner of the line, p1, in tc seconds. \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits corner at %g, %g in %g\\n", 37, p1.x, p1.y, t);

}

(\*timeOfCollision) = t;

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_corner, p1); /\* point no, lineC. \*/

if (drawPrediction) /\* point no, lineC. \*/

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (center, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugCircle (p1, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

/\* ii \*/

if (earlierCircleCollision ((\*edesc), cid, pid, &t, (\*timeOfCollision), &cp, p2.x, center.x, pvx, cvx, pax, cax, p2.y, center.y, pvy, cvy, pay, cay, radius, 0.0))

{

/\* circle hits corner of the line, p2, in tc seconds. \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits corner at %g, %g in %g (lineP+1)\\n", 48, p2.x, p2.y, t);

}

(\*timeOfCollision) = t;

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP+1, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_corner, p2); /\* point no, lineP+1. \*/

if (drawPrediction) /\* point no, lineP+1. \*/

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (cPtr->c.pos, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugCircle (p2, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

velCircle = coord\_initCoord (cvx, cvy);

accelCircle = coord\_initCoord (cax, cay);

velLine = coord\_initCoord (pvx, pvy);

accelLine = coord\_initCoord (pax, pay);

/\* iii and iv \*/

findEarlierCircleEdgeCollision (timeOfCollision, cid, pid, lineP, lineC, edesc, center, radius, velCircle, accelCircle, p1, p2, velLine, accelLine, createDesc);

}

/\*

findCollisionCircleLineOrbiting - find the time (if any) between line number, lineP, in polygon, pPtr,

and the circle, cPtr. cPtr can also be a polygon in which case lineC

is the particular line under question. Line on line collision is broken

down into circle line calls which allows for code reuse.

Either cPtr or pPtr or both are orbiting.

\*/

static void findCollisionCircleLineOrbiting (Object cPtr, Object pPtr, unsigned int lineP, unsigned int lineC, coord\_Coord center, double radius, eventDesc \*edesc, double \*timeOfCollision, descP createDesc)

{

double t;

double cr;

double cw;

double pr;

double pw;

coord\_Coord cv;

coord\_Coord ca;

coord\_Coord ccofg;

coord\_Coord pv;

coord\_Coord pa;

coord\_Coord pcofg;

coord\_Coord p1;

coord\_Coord p2;

unsigned int cid;

unsigned int pid;

cid = cPtr->id;

pid = pPtr->id;

getPolygonLine (lineP, pPtr, &p1, &p2);

/\* we perform 4 checks.

(i) and (ii) pretend the circle has radius 0.0 and see if it hits two new circles at

point, p1, and, p2 with the original radius.

(iii) and (iv) now draw two lines between the edge of the two new circles and see if the

center of the original circle intersects with either line.

the smallest positive time is the time of the next collision.

\*/

getObjectValues (cPtr, &cv.x, &cv.y, &ca.x, &ca.y);

getObjectValues (pPtr, &pv.x, &pv.y, &pa.x, &pa.y);

getObjectOrbitingValues (cPtr, &cr, &cw, &ccofg);

getObjectOrbitingValues (pPtr, &pr, &pw, &pcofg);

/\* i \*/

if (earlierCircleCollisionOrbiting (&t, timeOfCollision, center, radius, cr, cw, ccofg, cv, ca, p1, 0.0, pr, pw, pcofg, pv, pa))

{

/\* circle hits corner of the line, p1, in tc seconds. \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits corner at %g, %g in %g seconds\\n", 45, p1.x, p1.y, t);

}

(\*timeOfCollision) = t;

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_corner, p1); /\* point no, lineC. \*/

if (drawPrediction) /\* point no, lineC. \*/

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (center, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugCircle (p1, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

/\* ii \*/

if (earlierCircleCollisionOrbiting (&t, timeOfCollision, center, radius, cr, cw, ccofg, cv, ca, p2, 0.0, pr, pw, pcofg, pv, pa))

{

/\* circle hits corner of the line, p2, in tc seconds. \*/

if (Debugging)

{

libc\_printf ((char \*) "circle hits corner at %g, %g in %g seconds (lineP+1)\\n", 56, p2.x, p2.y, t);

}

(\*timeOfCollision) = t;

(\*edesc) = (\*createDesc.proc) ((\*edesc), cid, pid, lineP+1, lineC, (history\_whereHit) history\_corner, (history\_whereHit) history\_corner, p2); /\* point no, lineP+1. \*/

if (drawPrediction) /\* point no, lineP+1. \*/

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugCircle (cPtr->c.pos, 0.02, (deviceIf\_Colour) deviceIf\_white ());

debugCircle (p2, 0.02, (deviceIf\_Colour) deviceIf\_white ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

}

}

/\*

findCollisionCirclePolygon - find the smallest positive time (if any) between the polygon and circle.

If a collision if found then, tc, is assigned to the time and cid, pid

are set to the circle id and polygon id respectively.

\*/

static void findCollisionCirclePolygon (Object cPtr, Object pPtr, eventDesc \*edesc, double \*tc)

{

unsigned int i;

Assert (cPtr->object == circleOb, 6095);

Assert (pPtr->object == polygonOb, 6096);

if (isOrbiting (pPtr))

{

for (i=1; i<=pPtr->p.nPoints; i++)

{

findCollisionCircleLineOrbiting (cPtr, pPtr, i, 0, cPtr->c.pos, cPtr->c.r, edesc, tc, (descP) {(descP\_t) makeCirclesPolygonDesc});

}

}

else

{

for (i=1; i<=pPtr->p.nPoints; i++)

{

findCollisionCircleLine (cPtr, pPtr, i, 0, cPtr->c.pos, cPtr->c.r, edesc, tc, (descP) {(descP\_t) makeCirclesPolygonDesc});

}

}

}

/\*

makePolygonPolygon - return a new eventDesc indicating that we have a polygon/polygon collision

event.

\*/

static eventDesc makePolygonPolygon (eventDesc edesc, unsigned int id1, unsigned int id2, unsigned int lineCorner2, unsigned int lineCorner1, history\_whereHit wpid1, history\_whereHit wpid2, coord\_Coord collisionPoint)

{

if (edesc == NULL)

{

edesc = newDesc ();

}

edesc->etype = polygonPolygonEvent;

edesc->pp.cPoint = collisionPoint;

edesc->pp.pid1 = id1;

edesc->pp.pid2 = id2;

edesc->pp.wpid1 = wpid1;

edesc->pp.wpid2 = wpid2;

edesc->pp.lineCorner1 = lineCorner1;

edesc->pp.lineCorner2 = lineCorner2;

if (PolygonDebugging)

{

dumpDesc (edesc);

}

return edesc;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isOrbiting - return TRUE if object, o, is rotating.

\*/

static unsigned int isOrbiting (Object o)

{

return ! (roots\_nearZero (o->angularVelocity));

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

findCollisionLineLine - find the smallest time in the future when two lines collide.

The event descriptor, edesc, will contain the description of the collision

and, tc, the time of collision in the future.

\*/

static void findCollisionLineLine (Object iPtr, Object jPtr, unsigned int jLine, unsigned int iLine, eventDesc \*edesc, double \*tc)

{

if ((isOrbiting (iPtr)) || (isOrbiting (jPtr)))

{

/\* findCollisionLineLineOrbiting (iPtr, jPtr, jLine, iLine, edesc, tc) \*/

return;

}

else

{

findCollisionLineLineNonOrbiting (iPtr, jPtr, jLine, iLine, edesc, tc);

}

}

/\*

findCollisionPointLine - determines whether point, p, which is at one of the ends of iPtr:iLine will hit

line jPtr:jLine. If so then the time of collision is recorded in, tc, (provided

it is sooner than the current value of tc). It also updates the event descriptor,

edesc.

\*/

static void findCollisionPointLine (Object iPtr, Object jPtr, unsigned int jLine, unsigned int iLine, coord\_Coord p, eventDesc \*edesc, double \*tc)

{

coord\_Coord i0;

coord\_Coord i1;

getPolygonLine (iLine, iPtr, &i0, &i1);

Assert ((coord\_equalCoord (i0, p)) || (coord\_equalCoord (i1, p)), 6206);

findCollisionCircleLine (iPtr, jPtr, jLine, iLine, p, 0.0, edesc, tc, (descP) {(descP\_t) makePolygonPolygon});

}

/\*

findCollisionPointLine - determines whether point, p, which is at one of the ends of iPtr:iLine will hit

line jPtr:jLine. If so then the time of collision is recorded in, tc, (provided

it is sooner than the current value of tc). It also updates the event descriptor,

edesc.

\*/

static void findCollisionLineLineNonOrbiting (Object iPtr, Object jPtr, unsigned int iLine, unsigned int jLine, eventDesc \*edesc, double \*tc)

{

coord\_Coord i0;

coord\_Coord i1;

coord\_Coord j0;

coord\_Coord j1;

getPolygonLine (iLine, iPtr, &i0, &i1);

getPolygonLine (jLine, jPtr, &j0, &j1);

if (FALSE)

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

debugLine (i0, i1, (deviceIf\_Colour) deviceIf\_white ());

debugLine (j0, j1, (deviceIf\_Colour) deviceIf\_yellow ());

deviceIf\_flipBuffer ();

GC\_collectAll ();

}

findCollisionPointLine (iPtr, jPtr, jLine, iLine, i0, edesc, tc); /\* i0 crossing jLine. \*/

findCollisionPointLine (iPtr, jPtr, jLine, iLine, i1, edesc, tc); /\* i1 crossing jLine. \*/

findCollisionPointLine (jPtr, iPtr, iLine, jLine, j0, edesc, tc); /\* j0 crossing iLine. \*/

findCollisionPointLine (jPtr, iPtr, iLine, jLine, j1, edesc, tc); /\* j1 crossing iLine. \*/

}

/\*

Abs - return the absolute value of, r.

\*/

static double Abs (double r)

{

if (r < 0.0)

{

return -r;

}

else

{

return r;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

findAllTimesOfCollisionRLineRPoint -

4 4 4

- ((16 j k - 16 d e ) t

3 3 3 3 3

+ (64 j k l - 32 %pi j k - 64 d e f + 32 %pi d e ) t

2 2 2 2 2 2 2

+ (96 j k l - 96 %pi j k l + (24 %pi - 32) j k + 32 i - 96 d e f

2 2 2 2

+ 96 %pi d e f + (32 - 24 %pi ) d e - 32 c) t

3 2 2 3

+ (64 j k l - 96 %pi j k l + (48 %pi - 64) j k l + (32 %pi - 8 %pi ) j k

3 2 2

+ 64 h - 64 d e f + 96 %pi d e f + (64 - 48 %pi ) d e f

3 4 3

+ (8 %pi - 32 %pi) d e - 64 b) t + 16 j l - 32 %pi j l

2 2 3 4 2

+ (24 %pi - 32) j l + (32 %pi - 8 %pi ) j l + (%pi - 8 %pi + 64) j + 64 g

4 3 2 2 3

- 16 d f + 32 %pi d f + (32 - 24 %pi ) d f + (8 %pi - 32 %pi) d f

4 2

+ (- %pi + 8 %pi - 64) d - 64 a)/64 = 0

\*/

static unsigned int findAllTimesOfCollisionRLineRPoint (double a, double b, double c, double d, double e, double f, double g, double h, double i, double j, double k, double l, double \*t, unsigned int \_t\_high)

{

double A;

double B;

double C;

double D;

double E;

A = ((16.0\*j)\*(quad (k)))-((16.0\*d)\*(quad (e)));

B = (((((64.0\*j)\*(cub (k)))\*l)-(((32.0\*MathLib0\_pi)\*j)\*(cub (k))))-(((64.0\*d)\*(cub (e)))\*f))+(((32.0\*MathLib0\_pi)\*d)\*(cub (e)));

C = (((((((((96.0\*j)\*(sqr (k)))\*(sqr (l)))-((((96.0\*MathLib0\_pi)\*j)\*(sqr (k)))\*l))+((((24.0\*(sqr (MathLib0\_pi)))-32.0)\*j)\*(sqr (k))))+(32.0\*i))-(((96.0\*d)\*(sqr (e)))\*(sqr (f))))+((((96.0\*MathLib0\_pi)\*d)\*(sqr (e)))\*f))+(((32.0-(24.0\*(sqr (MathLib0\_pi))))\*d)\*(sqr (e))))-(32.0\*c);

D = (((((((((((64.0\*j)\*k)\*(cub (l)))-((((96.0\*MathLib0\_pi)\*j)\*k)\*(sqr (l))))+(((((48.0\*(sqr (MathLib0\_pi)))-64.0)\*j)\*k)\*l))+((((32.0\*MathLib0\_pi)-(8.0\*(sqr (MathLib0\_pi))))\*j)\*k))+(64.0\*h))-(((64.0\*d)\*e)\*(cub (f))))+((((96.0\*MathLib0\_pi)\*d)\*e)\*(sqr (f))))+((((64.0-(48.0\*(sqr (MathLib0\_pi))))\*d)\*e)\*f))+((((8.0\*(cub (MathLib0\_pi)))-(32.0\*MathLib0\_pi))\*d)\*e))-(64.0\*b);

E = ((((((((((((16.0\*j)\*(quad (l)))-(((32.0\*MathLib0\_pi)\*j)\*(cub (l))))+((((24.0\*(sqr (MathLib0\_pi)))-32.0)\*j)\*(sqr (l))))+((((32.0\*MathLib0\_pi)-(8.0\*(cub (MathLib0\_pi))))\*j)\*l))+((((cub (MathLib0\_pi))-(8.0\*(sqr (MathLib0\_pi))))+64.0)\*j))+(64.0\*g))-((16.0\*d)\*(quad (f))))+(((32.0\*MathLib0\_pi)\*d)\*(cub (f))))+(((32.0-(24.0\*(sqr (MathLib0\_pi))))\*d)\*(sqr (f))))+((((8.0\*(cub (MathLib0\_pi)))-(32.0\*MathLib0\_pi))\*d)\*f))+((((-(quad (MathLib0\_pi)))+(8.0\*(sqr (MathLib0\_pi))))-64.0)\*d))-(64.0\*a);

return roots\_findAllRootsQuartic (A, B, C, D, E, (double \*) t, \_t\_high);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

findEarliestCollisionRLineRPoint - find the earliest time when rotating point, j, collides with rotating line, i.

\*/

static void findEarliestCollisionRLineRPoint (Object iPtr, Object jPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc, double si, double ui, double ai, double ri, double wi, double oi, double sj, double uj, double aj, double rj, double wj, double oj, coord\_Coord p1, coord\_Coord p2)

{

typedef struct \_T16\_a \_T16;

struct \_T16\_a { double array[3+1]; };

\_T16 t;

unsigned int k;

unsigned int n;

double l;

}

/\*

findCollisionLineRPoint -

\*/

static void findCollisionLineRPoint (Object iPtr, Object rPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc)

{

polar\_Polar jp;

polar\_Polar pol1;

polar\_Polar pol2;

double o;

double offset;

double jw;

double iw;

coord\_Coord jpos;

coord\_Coord p1;

coord\_Coord p2;

coord\_Coord iu;

coord\_Coord ia;

coord\_Coord ju;

coord\_Coord ja;

coord\_Coord rcofg;

/\* now we ask when/if point jp crosses line p1, p2 \*/

findEarliestCollisionRLineRPoint (iPtr, rPtr, i, j, edesc, tc, p1.y, iu.y, ia.y, offset, iw, o, jpos.y, ju.y, ja.y, jp.r, jw, jp.w, p1, p2);

}

/\*

findCollisionLineRLine - find the time of collision between line, iPtr, and rotating line, rPtr.

\*/

static void findCollisionLineRLine (Object iPtr, Object rPtr, unsigned int i, unsigned int j, eventDesc \*edesc, double \*tc)

{

/\* test point rj-1 crossing line i \*/

findCollisionLineRPoint (iPtr, rPtr, i, j-1, edesc, tc);

/\* test point rj crossing line i \*/

findCollisionLineRPoint (iPtr, rPtr, i, j, edesc, tc);

/\* test point ii-1 crossing line j \*/

findCollisionLineRPoint (rPtr, iPtr, j, i-1, edesc, tc);

/\* test point ii crossing line j \*/

findCollisionLineRPoint (rPtr, iPtr, j, i, edesc, tc);

}

/\*

dumpLine -

\*/

static void dumpLine (unsigned int id, unsigned int line, history\_whereHit at)

{

coord\_Coord p1;

coord\_Coord p2;

getPolygonLine (line, (Object) Indexing\_GetIndice (objects, id), &p1, &p2);

libc\_printf ((char \*) "poly/line %d:%d ", 16, id, line);

dumpCoord (p1);

libc\_printf ((char \*) " ", 1);

dumpCoord (p2);

libc\_printf ((char \*) " ", 1);

dumpWhere (at);

libc\_printf ((char \*) "\\n", 2);

}

/\*

dumpWhere -

\*/

static void dumpWhere (history\_whereHit at)

{

switch (at)

{

case history\_edge:

libc\_printf ((char \*) "edge", 4);

break;

case history\_corner:

libc\_printf ((char \*) "corner", 6);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

dumpDesc -

\*/

static void dumpDesc (eventDesc e)

{

if (e != NULL)

{

switch (e->etype)

{

case frameEvent:

break;

case circlesEvent:

libc\_printf ((char \*) "circle/circle (%d:%d) at point ", 32, e->cc.cid1, e->cc.cid2);

dumpCoord (e->cc.cPoint);

libc\_printf ((char \*) "\\n", 2);

break;

case circlePolygonEvent:

libc\_printf ((char \*) "circle/polygon (%d:%d) at point ", 33, e->cp.pid, e->cp.cid);

dumpCoord (e->cp.cPoint);

libc\_printf ((char \*) " line %d, pointno %d\\n", 22, e->cp.lineNo, e->cp.pointNo);

break;

case polygonPolygonEvent:

libc\_printf ((char \*) "polygon/polygon (%d:%d) (%d:%d) at point\\n", 43, e->pp.pid1, e->pp.lineCorner1, e->pp.pid2, e->pp.lineCorner2);

dumpCoord (e->pp.cPoint);

libc\_printf ((char \*) " ", 2);

dumpLine (e->pp.pid1, e->pp.lineCorner1, e->pp.wpid1);

libc\_printf ((char \*) " ", 2);

dumpLine (e->pp.pid2, e->pp.lineCorner2, e->pp.wpid2);

libc\_printf ((char \*) " collision ", 12);

dumpCoord (e->pp.cPoint);

libc\_printf ((char \*) "\\n", 2);

break;

default:

break;

}

}

}

/\*

findCollisionPolygonPolygon - find the smallest positive time (if any) between the polygons, iPtr

and jPtr colliding.

If a collision if found then, tc, is assigned to the time and the

event descriptor is filled in.

\*/

static void findCollisionPolygonPolygon (Object iPtr, Object jPtr, eventDesc \*edesc, double \*tc)

{

unsigned int i;

unsigned int j;

Assert (iPtr != jPtr, 6502);

i = 1;

while (i <= iPtr->p.nPoints)

{

j = 1;

while (j <= jPtr->p.nPoints)

{

findCollisionLineLine (iPtr, jPtr, i, j, edesc, tc);

j += 1;

}

i += 1;

}

}

/\*

findCollisionPolygonRPolygon - find the smallest positive time (if any) between the polygons, iPtr

and rPtr colliding.

rPtr is a rotating polygon and iPtr is not rotating.

If a collision if found then, tc, is assigned to the time and the

event descriptor is filled in.

We check possible collision times between all lines of both polygons,

we separate out the rotating polygon from non rotating polygon

as the collision equations only generate a polynomial order 4 rather

than order 8 if both are rotating.

\*/

static void findCollisionPolygonRPolygon (Object iPtr, Object rPtr, eventDesc \*edesc, double \*tc)

{

unsigned int i;

unsigned int j;

}

/\*

findCollision -

\*/

static void findCollision (Object iptr, Object jptr, eventDesc \*edesc, double \*tc)

{

if (trace)

{

libc\_printf ((char \*) "findCollision entered and best time is %g\\n", 43, (\*tc));

}

if (! (iptr->fixed && jptr->fixed))

{

/\* avoid gcc warning by using compound statement even if not strictly necessary. \*/

if ((iptr->object == circleOb) && (jptr->object == circleOb))

{

/\* avoid dangling else. \*/

findCollisionCircles (iptr, jptr, edesc, tc);

if (trace)

{

libc\_printf ((char \*) "findCollision (circles) best time is %g\\n", 41, (\*tc));

}

}

else if ((iptr->object == circleOb) && (jptr->object == polygonOb))

{

/\* avoid dangling else. \*/

findCollisionCirclePolygon (iptr, jptr, edesc, tc);

}

else if ((iptr->object == polygonOb) && (jptr->object == circleOb))

{

/\* avoid dangling else. \*/

findCollisionCirclePolygon (jptr, iptr, edesc, tc);

}

else if ((iptr->object == polygonOb) && (jptr->object == polygonOb))

{

/\* avoid dangling else. \*/

findCollisionPolygonPolygon (jptr, iptr, edesc, tc);

}

}

if (trace)

{

libc\_printf ((char \*) "findCollision exiting and best time is %g\\n", 43, (\*tc));

}

}

/\*

debugFrame - debug frame at time, e.

\*/

static void debugFrame (eventQueue e)

{

drawBackground ((deviceIf\_Colour) deviceIf\_yellow ());

drawFrame ((eventQueue) NULL);

}

/\*

addDebugging - add a debugging event at time, t.

\*/

static void addDebugging (double t, eventDesc edesc)

{

eventQueue e;

e = newEvent ();

e->time\_ = t;

e->p.proc = debugFrame;

e->ePtr = edesc;

e->next = NULL;

addRelative (e);

}

/\*

anticipateCollision - stores the collision in the anticipated list.

\*/

static void anticipateCollision (double tc, eventDesc edesc)

{

switch (edesc->etype)

{

case circlesEvent:

history\_anticipateC (currentTime+tc, edesc->cc.cid1, edesc->cc.cid2, edesc->cc.cPoint);

break;

case circlePolygonEvent:

history\_anticipateC (currentTime+tc, edesc->cp.pid, edesc->cp.cid, edesc->cp.cPoint);

break;

case polygonPolygonEvent:

history\_anticipateC (currentTime+tc, edesc->pp.pid1, edesc->pp.pid2, edesc->pp.cPoint);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

collisionOccurred - stores the collision in the history list.

\*/

static void collisionOccurred (eventDesc edesc)

{

switch (edesc->etype)

{

case circlesEvent:

history\_occurredC (currentTime, edesc->cc.cid1, edesc->cc.cid2, edesc->cc.cPoint);

break;

case circlePolygonEvent:

history\_occurredC (currentTime, edesc->cp.pid, edesc->cp.cid, edesc->cp.cPoint);

break;

case polygonPolygonEvent:

history\_occurredC (currentTime, edesc->pp.pid1, edesc->pp.pid2, edesc->pp.cPoint);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

subEvent - remove event, e, from the relative time ordered event queue.

\*/

static void subEvent (eventQueue e)

{

eventQueue before;

eventQueue f;

f = eventQ;

before = NULL;

while ((f != e) && (f != NULL))

{

before = f;

f = f->next;

}

if (f != NULL)

{

Assert (f == e, 6665);

if (before == NULL)

{

/\* avoid dangling else. \*/

Assert (eventQ == f, 6668);

Assert (eventQ == e, 6669);

eventQ = eventQ->next;

if (eventQ != NULL)

{

eventQ->time\_ = eventQ->time\_+e->time\_;

}

}

else

{

before->next = e->next;

if (e->next != NULL)

{

e->next->time\_ = e->next->time\_+e->time\_;

}

}

disposeEvent (e);

}

}

/\*

removeCollisionEvent -

\*/

static void removeCollisionEvent (void)

{

eventQueue e;

if (trace)

{

libc\_printf ((char \*) "removeCollisionEvent\\n", 22);

}

e = eventQ;

while (e != NULL)

{

if (e->kind == collisionKind)

{

subEvent (e);

if (trace)

{

libc\_printf ((char \*) "return removeCollisionEvent\\n", 29);

}

return;

}

else

{

e = e->next;

}

}

if (trace)

{

libc\_printf ((char \*) "exit removeCollisionEvent\\n", 27);

}

}

/\*

removeSpringEvents - removes all spring events.

\*/

static void removeSpringEvents (void)

{

eventQueue e;

e = eventQ;

while (e != NULL)

{

if (e->kind == springKind)

{

subEvent (e);

e = eventQ;

}

else

{

e = e->next;

}

}

}

/\*

getSpringEndValues - it retrieves the:

CofG : c

velocity : v

acceleration: a

of object, o.

\*/

static void getSpringEndValues (unsigned int o, coord\_Coord \*c, coord\_Coord \*v, coord\_Coord \*a)

{

Object ptr;

ptr = Indexing\_GetIndice (objects, o);

(\*c) = getCofG (o);

(\*v) = getVelCoord (ptr);

(\*a) = getAccelCoord (ptr);

}

/\*

manualCircleCollision -

\*/

static void manualCircleCollision (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h, double k, double l, double m, double n, double o, double p)

{

if (trace)

{

libc\_printf ((char \*) "circle 1 pos: %g, %g\\n", 23, a, g);

libc\_printf ((char \*) " vel: %g, %g\\n", 23, c, k);

libc\_printf ((char \*) " acc: %g, %g\\n", 23, e, m);

libc\_printf ((char \*) "circle 2 pos: %g, %g\\n", 23, b, h);

libc\_printf ((char \*) " vel: %g, %g\\n", 23, d, l);

libc\_printf ((char \*) " acc: %g, %g\\n", 23, f, n);

libc\_printf ((char \*) " total distance of %g\\n", 24, o+p);

}

array[4] = (((((sqr (n))-((2.0\*m)\*n))+(sqr (m)))+(sqr (f)))-((2.0\*e)\*f))+(sqr (e));

array[3] = (((((4.0\*l)-(4.0\*k))\*n)+(((4.0\*k)-(4.0\*l))\*m))+(((4.0\*d)-(4.0\*c))\*f))+(((4.0\*c)-(4.0\*d))\*e);

array[2] = (((((((((((4.0\*h)-(4.0\*g))\*n)+(((4.0\*g)-(4.0\*h))\*m))+(4.0\*(sqr (l))))-((8.0\*k)\*l))+(4.0\*(sqr (k))))+(((4.0\*b)-(4.0\*a))\*f))+(((4.0\*a)-(4.0\*b))\*e))+(4.0\*(sqr (d))))-((8.0\*c)\*d))+(4.0\*(sqr (c)));

array[1] = (((((8.0\*h)-(8.0\*g))\*l)+(((8.0\*g)-(8.0\*h))\*k))+(((8.0\*b)-(8.0\*a))\*d))+(((8.0\*a)-(8.0\*b))\*c);

array[0] = ((((((4.0\*(sqr (h)))-((8.0\*g)\*h))+(4.0\*(sqr (g))))+(4.0\*(sqr (b))))-((8.0\*a)\*b))+(4.0\*(sqr (a))))-(sqr (2.0\*(p+o)));

}

/\*

earlierSpringLength - i is a spring.

c1, c2 are the c of g of the objects 1 and 2.

v1, v2 are the velocities of objects 1 and 2.

a1, a2 are the acceleration of objects 1 and 2.

Single letter variables are used since wxmaxima

only operates with these. Thus the output from wxmaxima

can be cut and pasted into the program.

a = c1.x

b = c1.y

c = v1.x

d = v1.y

e = a1.x

f = a1.y

g = c2.x

h = c2.y

k = v2.x

l = v2.y

m = a2.x

n = a2.y

t is the time of this collision (if any)

bestTimeOfCollision is earlier known collision so far.

\*/

static unsigned int earlierSpringLength (eventDesc edesc, unsigned int id, double \*t, double bestTime, coord\_Coord c1, coord\_Coord v1, coord\_Coord a1, coord\_Coord c2, coord\_Coord v2, coord\_Coord a2, double l, history\_springPoint sp)

{

typedef struct \_T17\_a \_T17;

typedef struct \_T18\_a \_T18;

struct \_T17\_a { double array[4+1]; };

struct \_T18\_a { double array[3+1]; };

\_T17 array;

\_T18 roots;

unsigned int n;

unsigned int j;

double T;

manualCircleCollision ((double \*) &array.array[0], 4, c1.x, c2.x, v1.x, v2.x, a1.x, a2.x, c1.y, c2.y, v1.y, v2.y, a1.y, a2.y, l, 0.0);

/\* now solve for values of t which satisfy array[4]\*t^4 + array[3]\*t^3 + array[2]\*t^2 + array[1]\*t^1 + array[0]\*t^0 = 0 \*/

n = roots\_findQuarticRoots (array.array[4], array.array[3], array.array[2], array.array[1], array.array[0], (double \*) &roots.array[0], 3);

j = 0;

/\* we try each root in turn, selecting the smallest positive which has not been seen before. \*/

while (j < n)

{

(\*t) = roots.array[j];

T = ((((array.array[4]\*((sqr ((\*t)))\*(sqr ((\*t)))))+(array.array[3]\*((sqr ((\*t)))\*(\*t))))+(array.array[2]\*(sqr ((\*t)))))+(array.array[1]\*(\*t)))+array.array[0];

if (Debugging)

{

libc\_printf ((char \*) "%gt^4 + %gt^3 +%gt^2 + %gt + %g = %g (t=%g)\\n", 48, array.array[4], array.array[3], array.array[2], array.array[1], array.array[0], T, (\*t));

libc\_printf ((char \*) "found spring reaches length %g at %g\\n", 38, l, (\*t));

}

Assert ((\*t) >= 0.0, 6842);

/\* remember edesc = NIL if bestTime is unassigned. \*/

if ((edesc == NULL) || ((\*t) < bestTime))

{

if (! (history\_isDuplicateS (currentTime, (\*t), id, sp)))

{

/\* ok, this has not been seen before. \*/

return TRUE;

}

}

j += 1;

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

makeSpringDesc - creates and fills in the spring descriptor.

\*/

static eventDesc makeSpringDesc (eventDesc edesc, unsigned int i, history\_springPoint stype)

{

if (edesc == NULL)

{

edesc = newDesc ();

}

edesc->etype = springEvent;

edesc->sp.id = i;

edesc->sp.type = stype;

return edesc;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

calcSpringLengthEvents -

\*/

static void calcSpringLengthEvents (unsigned int i)

{

eventDesc edesc;

double a;

double b;

double c;

double d;

double e;

double f;

double g;

double h;

double k;

double l;

double m;

double n;

double o;

double p;

coord\_Coord cp;

coord\_Coord c1;

coord\_Coord v1;

coord\_Coord a1;

coord\_Coord c2;

coord\_Coord v2;

coord\_Coord a2;

double test;

double ts;

double t;

unsigned int id1;

unsigned int id2;

Object iptr;

Object id1ptr;

Object id2ptr;

Assert (isSpringObject (i), 6892);

iptr = Indexing\_GetIndice (objects, i);

id1 = iptr->s.id1;

id2 = iptr->s.id2;

ts = -1.0;

edesc = NULL;

t = -1.0;

getSpringEndValues (id1, &c1, &v1, &a1);

getSpringEndValues (id2, &c2, &v2, &a2);

/\*

test := -1.0 ;

IF earlierCircleCollision (edesc, i, i,

t, ts, cp,

c1.x, c2.x, v1.x, v2.x, a1.x, a2.x,

c1.y, c2.y, v1.y, v2.y, a1.y, a2.y, iptr^.s.l0, 0.0)

THEN

test := t ;

edesc := NIL ;

ts := -1.0 ;

printf ("should get to the mid point in %g seconds

", t)

END ;

t := -1.0 ;

id1ptr := GetIndice (objects, id1) ;

id2ptr := GetIndice (objects, id2) ;

getCircleValues (id1ptr, a, g, o, c, k, e, m) ;

b xj

h yj

p rj

d vxj

l vyj

f ajx

n ajy

getCircleValues (id2ptr, b, h, p, d, l, f, n) ;

o := 0.0 ;

p := iptr^.s.l0 ;

IF earlierCircleCollision (edesc, id1, id2,

t, ts, cp,

a, b, c, d, e, f, g, h, k, l, m, n, o, p)

THEN

test := t ;

edesc := NIL ;

ts := -1.0 ;

printf ("should get to the mid point in %g seconds

", t)

END ;

t := -1.0 ;

\*/

if (earlierSpringLength (edesc, i, &t, ts, c1, v1, a1, c2, v2, a2, iptr->s.l0, (history\_springPoint) history\_midPoint))

{

/\*

printf ("actually found a mid point value of %g

", t) ;

AssertRFail (test, t) ;

\*/

ts = t;

if (trace)

{

libc\_printf ((char \*) "spring %d reaches midpoint in %g seconds\\n", 42, i, t);

}

edesc = makeSpringDesc (edesc, i, (history\_springPoint) history\_midPoint);

addSpringEvent (ts, (eventProc) {(eventProc\_t) doSpring}, edesc);

anticipateSpring (ts, edesc);

}

if (iptr->s.hasCallBackLength)

{

t = -1.0;

edesc = NULL;

getSpringEndValues (id1, &c1, &v1, &a1);

getSpringEndValues (id2, &c2, &v2, &a2);

if (earlierSpringLength (edesc, i, &t, ts, c1, v1, a1, c2, v2, a2, iptr->s.cbl, (history\_springPoint) history\_callPoint))

{

ts = t;

if (trace)

{

libc\_printf ((char \*) "spring %d reaches the call length in %g seconds\\n", 49, i, t);

}

edesc = makeSpringDesc (edesc, i, (history\_springPoint) history\_callPoint);

addSpringEvent (ts, (eventProc) {(eventProc\_t) doSpring}, edesc);

anticipateSpring (ts, edesc);

if (trace)

{

printQueue ();

}

}

}

}

/\*

manualSpringVelocityZero -

\*/

static void manualSpringVelocityZero (double \*array, unsigned int \_array\_high, double a, double b, double c, double d, double e, double f, double g, double h)

{

/\* thanks to wxmaxima and max2code. \*/

array[0] = ((((0.0+0.0)-(sqr (g)))-(sqr (e)))+(sqr (c)))+(sqr (a));

array[1] = (((-((2.0\*g)\*h))-((2.0\*e)\*f))+((2.0\*c)\*d))+((2.0\*a)\*b);

array[2] = (((-(sqr (h)))-(sqr (f)))+(sqr (d)))+(sqr (b));

}

/\*

earlierSpringEnd - records the earliest time in the future when the

relative velocity between the two bodies is zero.

\*/

static void earlierSpringEnd (unsigned int id, coord\_Coord v1, coord\_Coord a1, coord\_Coord v2, coord\_Coord a2)

{

typedef struct \_T19\_a \_T19;

typedef struct \_T20\_a \_T20;

struct \_T19\_a { double array[2+1]; };

struct \_T20\_a { double array[1+1]; };

eventDesc edesc;

double t;

\_T19 array;

\_T20 root;

unsigned int n;

unsigned int i;

manualSpringVelocityZero ((double \*) &array.array[0], 2, v1.x, a1.x, v1.y, a1.y, v2.x, a2.x, v2.y, a2.y);

n = roots\_findQuadraticRoots (array.array[2], array.array[1], array.array[0], (double \*) &root.array[0], 1);

/\* now try each root in turn recording the lowest unique only. \*/

i = 0;

while (i < n)

{

t = root.array[i];

if (! (history\_isDuplicateS (currentTime, t, id, (history\_springPoint) history\_endPoint)))

{

/\* ok, this has not been seen before, so we add it. \*/

edesc = makeSpringDesc ((eventDesc) NULL, id, (history\_springPoint) history\_endPoint);

addSpringEvent (t, (eventProc) {(eventProc\_t) doSpring}, edesc);

anticipateSpring (t, edesc);

if (trace)

{

libc\_printf ((char \*) "end point calculated at time %g\\n", 33, t);

printQueue ();

}

}

i += 1;

}

}

/\*

calcSpringEndEvents - the spring reaches maximum compression or extension when the

relative velocity between the objects attached to the spring

is zero.

\*/

static void calcSpringEndEvents (unsigned int i)

{

coord\_Coord c1;

coord\_Coord v1;

coord\_Coord a1;

coord\_Coord c2;

coord\_Coord v2;

coord\_Coord a2;

double t;

unsigned int id1;

unsigned int id2;

Object iptr;

Object id1ptr;

Object id2ptr;

Assert (isSpringObject (i), 7054);

iptr = Indexing\_GetIndice (objects, i);

id1 = iptr->s.id1;

id2 = iptr->s.id2;

getSpringEndValues (id1, &c1, &v1, &a1);

getSpringEndValues (id2, &c2, &v2, &a2);

earlierSpringEnd (i, v1, a1, v2, a2);

}

/\*

calcSpringEndEventsKE - calcalate the time at which there is no Kinetic energy

in spring, i. This will be when the spring reaches

its end point.

\*/

static void calcSpringEndEventsKE (double \*ts, unsigned int i, eventDesc \*edesc)

{

coord\_Coord c1;

coord\_Coord v1;

coord\_Coord a1;

coord\_Coord c2;

coord\_Coord v2;

coord\_Coord a2;

double t;

double l1;

double d;

unsigned int id1;

unsigned int id2;

Object iptr;

Object id1ptr;

Object id2ptr;

}

/\*

calcSpringEventTime - calculates the time in the future when spring, i,

reaches its:

mid point and minimum or maximum extension.

Both events are stored as they may be independent.

If they are not independent, it wont matter as the

event queue will be recreated.

\*/

static void calcSpringEventTime (unsigned int i)

{

/\* gdbif.sleepSpin ; \*/

calcSpringLengthEvents (i);

if (DebugTrace)

{

printQueue ();

}

/\* these spring event categories, midEvent and lengthEvent are

treated independently and both stored on the queue. \*/

calcSpringEndEvents (i);

if (DebugTrace)

{

printQueue ();

}

/\*

calcSpringEndEventsKE (ts, i, edesc) ;

\*/

}

/\*

addSpringEvent -

\*/

static void addSpringEvent (double t, eventProc dop, eventDesc edesc)

{

eventQueue e;

if (Debugging)

{

libc\_printf ((char \*) "spring event will occur in %g simulated seconds\\n", 49, t);

}

Assert (t >= 0.0, 7159);

e = newEvent ();

e->kind = springKind;

e->time\_ = t;

e->p = dop;

e->ePtr = edesc;

e->next = NULL;

if (Debugging)

{

libc\_printf ((char \*) "spring event about to be added to this queue at %g in the future\\n", 66, t);

printQueue ();

}

addRelative (e);

if (Debugging)

{

libc\_printf ((char \*) "spring event has been added to this queue at %g in the future\\n", 63, t);

printQueue ();

}

}

/\*

reverseSpringAccel -

\*/

static void reverseSpringAccel (Object o)

{

Object id1p;

Object id2p;

Assert (isSpringObject (o->id), 7190);

if (! o->deleted && ! o->fixed)

{

id1p = Indexing\_GetIndice (objects, o->s.id1);

id2p = Indexing\_GetIndice (objects, o->s.id2);

/\* gdbif.sleepSpin ; \*/

if (trace)

{

libc\_printf ((char \*) "entered reverse spring acceleration\\n", 37);

dumpObject (o);

dumpObject (id1p);

dumpObject (id2p);

libc\_printf ((char \*) "reversing spring acceleration\\n", 31);

}

id1p->forceVec = coord\_subCoord (id1p->forceVec, o->s.f1);

id2p->forceVec = coord\_subCoord (id2p->forceVec, o->s.f2);

o->s.f1 = coord\_negateCoord (o->s.f1);

o->s.f2 = coord\_negateCoord (o->s.f2);

id1p->forceVec = coord\_addCoord (id1p->forceVec, o->s.f1);

id2p->forceVec = coord\_addCoord (id2p->forceVec, o->s.f2);

if (trace)

{

dumpObject (o);

dumpObject (id1p);

dumpObject (id2p);

libc\_printf ((char \*) "applying new force values\\n", 27);

}

applyForce ();

if (trace)

{

dumpObject (o);

dumpObject (id1p);

dumpObject (id2p);

libc\_printf ((char \*) "completed reverse acceleration\\n", 32);

}

}

}

/\*

zeroSpringAccel - assign (0.0, 0.0) to the acceration and force vectors.

\*/

static void zeroSpringAccel (Object o)

{

if (((o->object == springOb) && ! o->deleted) && ! o->fixed)

{

if (trace)

{

libc\_printf ((char \*) "zero spring acceleration\\n", 26);

}

o->s.a1 = coord\_initCoord (0.0, 0.0);

o->s.a2 = coord\_initCoord (0.0, 0.0);

o->s.f1 = coord\_initCoord (0.0, 0.0);

o->s.f2 = coord\_initCoord (0.0, 0.0);

}

}

/\*

doSpringMidPoint - reached the mid point of the spring, reverse the

acceleration of the sprung objects.

\*/

static void doSpringMidPoint (eventQueue e)

{

Object idptr;

Object id1ptr;

Object id2ptr;

if (trace)

{

libc\_printf ((char \*) "doSpringMidPoint called at time %g\\n", 36, currentTime);

}

/\* firstly we remove some energy from the moving objects. \*/

idptr = Indexing\_GetIndice (objects, e->ePtr->sp.id);

id1ptr = Indexing\_GetIndice (objects, idptr->s.id1);

id2ptr = Indexing\_GetIndice (objects, idptr->s.id2);

/\* Assert (nearSame (idptr^.s.l0, lengthCoord (subCoord (getCofG (idptr^.s.id1), getCofG (idptr^.s.id2)))), \_\_LINE\_\_) ; \*/

if (idptr->s.drawMid)

{

if (trace)

{

libc\_printf ((char \*) "about to draw spring mid frame\\n", 32);

}

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

if (trace)

{

libc\_printf ((char \*) "drawing mid points of the spring\\n", 34);

}

doDrawFrame (id1ptr, 0.0, (deviceIf\_Colour) idptr->s.midColour);

doDrawFrame (id2ptr, 0.0, (deviceIf\_Colour) idptr->s.midColour);

deviceIf\_flipBuffer ();

}

/\*

IF NOT id1ptr^.fixed

THEN

inElasticSpring (id1ptr^.vx) ;

inElasticSpring (id1ptr^.vy) ;

id1ptr^.vx := 0.0 ;

id1ptr^.vy := 0.0 ;

dumpObject (id1ptr)

END ;

IF NOT id2ptr^.fixed

THEN

inElasticSpring (id2ptr^.vx) ;

inElasticSpring (id2ptr^.vy) ;

id2ptr^.vx := 0.0 ;

id2ptr^.vy := 0.0 ;

dumpObject (id2ptr)

END ;

\*/

if (! FrameSprings)

{

if (trace)

{

dumpObject (idptr);

}

reverseSpringAccel (idptr);

if (trace)

{

dumpObject (id1ptr);

dumpObject (id2ptr);

}

}

if (trace)

{

libc\_printf ((char \*) "doSpringMidPoint finishing\\n", 28);

}

}

/\*

doSpringEndPoint - reached the end point of the spring, we

remove some energy from the sprung objects.

\*/

static void doSpringEndPoint (eventQueue e)

{

Object idptr;

Object id1ptr;

Object id2ptr;

idptr = Indexing\_GetIndice (objects, e->ePtr->sp.id);

id1ptr = Indexing\_GetIndice (objects, idptr->s.id1);

id2ptr = Indexing\_GetIndice (objects, idptr->s.id2);

if (idptr->s.drawEnd)

{

if (trace)

{

libc\_printf ((char \*) "about to draw spring end frame\\n", 32);

}

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

if (trace)

{

libc\_printf ((char \*) "drawing end points of the spring\\n", 34);

}

doDrawFrame (id1ptr, 0.0, (deviceIf\_Colour) idptr->s.endColour);

doDrawFrame (id2ptr, 0.0, (deviceIf\_Colour) idptr->s.endColour);

deviceIf\_flipBuffer ();

}

if (! FrameSprings)

{

/\* we remove some energy from the moving objects. \*/

recalculateForceEnergy ();

inElasticSpring (&id1ptr->saccel.x);

inElasticSpring (&id1ptr->saccel.y);

inElasticSpring (&id2ptr->saccel.x);

inElasticSpring (&id2ptr->saccel.y);

}

}

/\*

doSpringCallPoint - reached the user defined call point.

We need to activate the call back.

\*/

static void doSpringCallPoint (eventQueue e)

{

Object o;

fcDesc fc;

/\* gdbif.sleepSpin ; \*/

Assert (e->ePtr->etype == springEvent, 7394);

Assert (e->ePtr->sp.type == history\_callPoint, 7395);

o = Indexing\_GetIndice (objects, e->ePtr->sp.id);

Assert (isSpringObject (e->ePtr->sp.id), 7397);

o->s.hasCallBackLength = FALSE; /\* turn this off. \*/

/\* now invoke the call. \*/

twoDsim\_createFunctionEvent (0.0, o->s.func, o->id);

if (trace)

{

printQueue ();

}

}

/\*

doSpring - called whenever a spring event is processed.

\*/

static void doSpring (eventQueue e)

{

/\* gdbif.sleepSpin () ; \*/

if (trace)

{

libc\_printf ((char \*) "doSpring called\\n", 17);

if (e->ePtr->sp.type == history\_midPoint)

{} /\* empty. \*/

/\* gdbif.sleepSpin \*/

if (e->ePtr->sp.type == history\_endPoint)

{} /\* empty. \*/

/\* gdbif.sleepSpin \*/

}

updatePhysics ((e->ePtr->sp.type == history\_endPoint) || FrameSprings);

springOccurred (e->ePtr);

switch (e->ePtr->sp.type)

{

case history\_midPoint:

doSpringMidPoint (e);

break;

case history\_endPoint:

doSpringEndPoint (e);

break;

case history\_callPoint:

doSpringCallPoint (e);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\* gdbif.sleepSpin ; \*/

addNextObjectEvent ();

}

/\*

springOccurred - stores the spring event in the history list.

\*/

static void springOccurred (eventDesc edesc)

{

switch (edesc->etype)

{

case springEvent:

history\_occurredS (currentTime, edesc->sp.id, edesc->sp.type);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

anticipateSpring - stores the collision in the anticipated list.

\*/

static void anticipateSpring (double tc, eventDesc edesc)

{

switch (edesc->etype)

{

case springEvent:

history\_anticipateS (currentTime+tc, edesc->sp.id, edesc->sp.type);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

/\*

addNextSpringEvent -

\*/

static void addNextSpringEvent (void)

{

unsigned int n;

unsigned int i;

Object iptr;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

if (isSpringObject (i))

{

calcSpringEventTime (i);

}

i += 1;

}

}

/\*

optPredictiveBroadphase - this function returns the list after culling

any pair objects which are moving in opposite

directions. It should check for velocity and

acceleration - making sure that both have the

same sign.

\*/

static broadphase optPredictiveBroadphase (broadphase list)

{

/\* --fixme-- your code goes here. mcomp. \*/

return list;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

addNextCollisionEvent - recalculate the next collision event time.

\*/

static void addNextCollisionEvent (void)

{

double tc;

double old;

Object o0;

Object o1;

eventDesc edesc;

broadphase list;

broadphase b;

edesc = NULL;

tc = -1.0;

list = optPredictiveBroadphase (initBroadphase ());

/\*

IF list # NIL

THEN

gdbif.sleepSpin

END ;

\*/

b = list;

while (b != NULL)

{

o0 = Indexing\_GetIndice (objects, b->o0);

o1 = Indexing\_GetIndice (objects, b->o1);

if (trace)

{

libc\_printf ((char \*) "\*\* checking pair %d, %d\\n", 25, b->o0, b->o1);

old = tc;

}

findCollision (o0, o1, &edesc, &tc);

if (trace && (old != tc))

{

libc\_printf ((char \*) "\*\* collision found between pair %d, %d at time %g\\n", 51, b->o0, b->o1, tc);

if (edesc != NULL)

{

dumpDesc (edesc);

}

}

b = b->next;

}

killBroadphase (&list);

if (edesc != NULL)

{

/\* avoid dangling else. \*/

addCollisionEvent (tc, (eventProc) {(eventProc\_t) doCollision}, edesc);

anticipateCollision (tc, edesc);

if (PolygonDebugging)

{

printQueue ();

libc\_printf ((char \*) "collision detected\\n", 20);

dumpDesc (edesc);

}

}

else if (trace || PolygonDebugging)

{

/\* avoid dangling else. \*/

libc\_printf ((char \*) "no more collisions found\\n", 26);

}

}

/\*

determineFrameBased - return TRUE if we need to use frame based collision detection.

\*/

static unsigned int determineFrameBased (void)

{

unsigned int n;

unsigned int i;

Object o;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

if ((isPolygon (i)) && (! (twoDsim\_isFixed (i))))

{

o = Indexing\_GetIndice (objects, i);

if (! (roots\_nearZero (o->angularVelocity)))

{

return TRUE;

}

}

i += 1;

}

return FALSE;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

doAddNextObjectEvent - removes the next spring and collision event and recalculates

the time of both events.

\*/

static void doAddNextObjectEvent (void)

{

removeCollisionEvent ();

removeSpringEvents ();

if (trace)

{

libc\_printf ((char \*) "no spring or collision events here\\n", 36);

printQueue ();

}

/\* addNextSpringEvent must be run before addNextCollisionEvent

as it will update the spring acceleration \*/

addNextSpringEvent ();

addNextCollisionEvent ();

if (trace)

{

libc\_printf ((char \*) "event queue created and it looks like this\\n", 44);

printQueue ();

}

}

/\*

addNextObjectEvent - check to see if we are using predictive event mode

and if so then add the next predictive collision event.

\*/

static void addNextObjectEvent (void)

{

doAddNextObjectEvent ();

}

/\*

skipFor - skip displaying any frames for, t, simulated seconds.

\*/

static void skipFor (double t)

{

twoDsim\_simulateFor (t);

}

/\*

resetQueue -

\*/

static void resetQueue (void)

{

eventQueue c;

eventQueue f;

eventQueue e;

c = NULL; /\* collision event \*/

f = NULL; /\* draw frame event \*/

e = eventQ; /\* draw frame event \*/

while ((e != NULL) && ((c == NULL) || (f == NULL)))

{

if ((e->kind == collisionKind) || (e->kind == springKind))

{

c = e;

}

else if (e->kind == frameKind)

{

/\* avoid dangling else. \*/

f = e;

}

e = e->next;

}

if (f == NULL)

{

addEvent ((eventKind) frameKind, (1.0/framesPerSecond)-(currentTime-lastDrawTime), (eventProc) {(eventProc\_t) drawFrameEvent});

}

if (c == NULL)

{

addNextObjectEvent ();

}

}

/\*

disposeEvent - returns the event to the free queue.

\*/

static void disposeEvent (eventQueue e)

{

disposeDesc (&e->ePtr);

e->next = freeEvents;

freeEvents = e;

}

/\*

disposeDesc - returns the event desc to the free queue.

\*/

static void disposeDesc (eventDesc \*d)

{

if ((\*d) != NULL)

{

(\*d)->next = freeDesc;

freeDesc = (\*d);

(\*d) = NULL;

}

}

/\*

newDesc - returns a new eventDesc.

\*/

static eventDesc newDesc (void)

{

eventDesc e;

if (freeDesc == NULL)

{

Storage\_ALLOCATE ((void \*\*) &e, sizeof (\_T1));

}

else

{

e = freeDesc;

freeDesc = freeDesc->next;

}

return e;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

newEvent - returns a new eventQueue.

\*/

static eventQueue newEvent (void)

{

eventQueue e;

if (freeEvents == NULL)

{

Storage\_ALLOCATE ((void \*\*) &e, sizeof (\_T4));

}

else

{

e = freeEvents;

freeEvents = freeEvents->next;

}

e->ePtr = NULL;

return e;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

makeCirclesDesc - return a eventDesc which describes two circles colliding.

\*/

static eventDesc makeCirclesDesc (eventDesc \*edesc, unsigned int cid1, unsigned int cid2, coord\_Coord cp)

{

if ((\*edesc) == NULL)

{

(\*edesc) = newDesc ();

}

(\*edesc)->etype = circlesEvent;

(\*edesc)->cc.cPoint = cp;

(\*edesc)->cc.cid1 = cid1;

(\*edesc)->cc.cid2 = cid2;

return (\*edesc);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

addRelative - adds event, e, into the relative event queue.

\*/

static void addRelative (eventQueue e)

{

eventQueue before;

eventQueue after;

if (Debugging)

{

libc\_printf ((char \*) "addRelative entered, event abs time = %g\\n", 42, e->time\_);

printQueue ();

}

if (eventQ == NULL)

{

eventQ = e;

}

else if (e->time\_ < eventQ->time\_)

{

/\* avoid dangling else. \*/

eventQ->time\_ = eventQ->time\_-e->time\_;

e->next = eventQ;

eventQ = e;

}

else

{

/\* avoid dangling else. \*/

if (Debugging)

{

printQueue ();

}

before = eventQ;

after = eventQ->next;

e->time\_ = e->time\_-before->time\_;

while ((after != NULL) && (after->time\_ < e->time\_))

{

before = after;

e->time\_ = e->time\_-before->time\_;

after = after->next;

}

if (after != NULL)

{

after->time\_ = after->time\_-e->time\_;

}

before->next = e;

e->next = after;

}

if (Debugging)

{

libc\_printf ((char \*) "addRelative changed queue to:\\n", 31);

printQueue ();

libc\_printf ((char \*) "addRelative finishing now\\n", 27);

}

}

/\*

addEvent - adds an event which has no collision associated with it.

Typically this is a debugging event or display frame event.

\*/

static void addEvent (eventKind k, double t, eventProc dop)

{

eventQueue e;

if (Debugging)

{

libc\_printf ((char \*) "new event will occur at time %g in the future\\n", 47, t);

}

Assert (t >= 0.0, 7940);

e = newEvent ();

e->kind = k;

e->time\_ = t;

e->p = dop;

e->ePtr = NULL;

e->next = NULL;

addRelative (e);

}

/\*

assertMovement -

\*/

static void assertMovement (unsigned int id1, unsigned int id2, char \*message\_, unsigned int \_message\_high)

{

char message[\_message\_high+1];

/\* make a local copy of each unbounded array. \*/

memcpy (message, message\_, \_message\_high+1);

if ((twoDsim\_isFixed (id1)) && (twoDsim\_isFixed (id2)))

{

libc\_printf ((char \*) "assert failed, ", 15);

libc\_printf ((char \*) message, \_message\_high);

libc\_printf ((char \*) ", pge should not be detecting a collision event between two fixed objects %d and %d\\n", 85, id1, id2);

}

}

/\*

assertCollisionEvent - assert that the collision event consists of two non fixed objects.

\*/

static void assertCollisionEvent (eventDesc edesc)

{

if (edesc != NULL)

{

switch (edesc->etype)

{

case frameEvent:

break;

case circlesEvent:

assertMovement (edesc->cc.cid1, edesc->cc.cid2, (char \*) "circles", 7);

break;

case circlePolygonEvent:

assertMovement (edesc->cp.cid, edesc->cp.pid, (char \*) "circle polygon", 14);

break;

case polygonPolygonEvent:

assertMovement (edesc->pp.pid1, edesc->pp.pid2, (char \*) "polygon polygon", 15);

break;

case functionEvent:

case springEvent:

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

}

/\*

addCollisionEvent - adds a collision event, the edesc is attached to the,

eventQueue, which is placed onto the eventQ.

\*/

static void addCollisionEvent (double t, eventProc dop, eventDesc edesc)

{

eventQueue e;

if (Debugging)

{

libc\_printf ((char \*) "collision will occur in %g simulated seconds\\n", 46, t);

}

Assert (t >= 0.0, 8006);

e = newEvent ();

e->kind = collisionKind;

e->time\_ = t;

e->p = dop;

e->ePtr = edesc;

e->next = NULL;

if (Debugging)

{

libc\_printf ((char \*) "collision about to be added to this queue at %g in the future\\n", 63, t);

printQueue ();

}

addRelative (e);

if (Debugging)

{

libc\_printf ((char \*) "collision has been added to this queue at %g in the future\\n", 60, t);

printQueue ();

}

assertCollisionEvent (edesc);

}

/\*

isEvent - return TRUE if the next event is of kind, k.

\*/

static unsigned int isEvent (eventKind k)

{

if (eventQ == NULL)

{

return FALSE;

}

else

{

return eventQ->kind == k;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

dumpTime -

\*/

static void dumpTime (void)

{

libc\_printf ((char \*) " absolute time is %f\\n", 23, currentTime);

libc\_printf ((char \*) " last update time is %f\\n", 26, lastUpdateTime);

}

/\*

recordEvent -

\*/

static void recordEvent (void)

{

checkOpened ();

if (eventQ != NULL)

{

if (DebugTrace)

{

libc\_printf ((char \*) "before writeEvent\\n", 19);

}

/\* gdbif.sleepSpin ; \*/

writeEvent (eventQ);

if (DebugTrace)

{

libc\_printf ((char \*) "after writeEvent\\n", 18);

}

}

}

/\*

pumpQueue - prime the event queue with initial frame and collision events.

\*/

static void pumpQueue (void)

{

resetQueue ();

recordEvent ();

}

/\*

up - recreate the event queue.

The pair up/down must be used to shutdown

the event queue if the world is to be altered.

\*/

static void up (void)

{

history\_forgetFuture ();

resetQueue ();

if (DebugTrace)

{

libc\_printf ((char \*) "exiting up\\n", 12);

printQueue ();

libc\_printf ((char \*) "remaking queue\\n", 16);

removeCollisionEvent ();

libc\_printf ((char \*) "collision event removed\\n", 25);

printQueue ();

libc\_printf ((char \*) "collision event added\\n", 23);

resetQueue ();

printQueue ();

libc\_printf ((char \*) "and again (collision event added)\\n", 35);

removeCollisionEvent ();

resetQueue ();

printQueue ();

libc\_printf ((char \*) "exiting finishing\\n", 19);

}

}

/\*

down - shutdown the event queue.

\*/

static void down (void)

{

if (DebugTrace)

{

libc\_printf ((char \*) "entered down\\n", 14);

printQueue ();

}

updatePhysics (TRUE);

removeCollisionEvent ();

}

/\*

killQueue - destroys the event queue and returns events to the free list.

\*/

static void killQueue (void)

{

eventQueue e;

if (eventQ != NULL)

{

e = eventQ;

while (e->next != NULL)

{

e = e->next;

}

e->next = freeEvents;

freeEvents = eventQ;

eventQ = NULL;

}

}

/\*

writeCircles -

\*/

static void writeCircles (cDesc c)

{

if (DebugTrace)

{

libc\_printf ((char \*) "writeCircleCircle %d %d\\n", 25, c.cid1, c.cid2);

}

NetworkOrder\_writeCoord (file, c.cPoint);

NetworkOrder\_writeCard (file, c.cid1);

NetworkOrder\_writeCard (file, c.cid2);

}

/\*

writeKind -

\*/

static void writeKind (history\_whereHit k)

{

NetworkOrder\_writeCard (file, ((unsigned int) (k)));

}

/\*

writeCirclePolygon -

\*/

static void writeCirclePolygon (cpDesc c)

{

if (DebugTrace)

{

libc\_printf ((char \*) "writeCirclePolygon %d %d\\n", 26, c.pid, c.cid);

}

NetworkOrder\_writeCoord (file, c.cPoint);

NetworkOrder\_writeCard (file, c.pid);

NetworkOrder\_writeCard (file, c.cid);

writeKind (c.wpid);

}

/\*

writePolygonPolygon -

\*/

static void writePolygonPolygon (ppDesc p)

{

if (DebugTrace)

{

libc\_printf ((char \*) "writePolygonPolygon %d %d\\n", 27, p.pid1, p.pid2);

}

NetworkOrder\_writeCoord (file, p.cPoint);

NetworkOrder\_writeCard (file, p.pid1);

NetworkOrder\_writeCard (file, p.pid2);

writeKind (p.wpid1);

writeKind (p.wpid2);

}

/\*

writeFunction -

\*/

static void writeFunction (fcDesc fc)

{

NetworkOrder\_writeCard (file, fc.id);

NetworkOrder\_writeCard (file, fc.param);

}

/\*

writeSpring -

\*/

static void writeSpring (spDesc sp)

{

NetworkOrder\_writeCard (file, sp.id);

NetworkOrder\_writeCard (file, ((unsigned int) (sp.type)));

}

/\*

writeDesc -

\*/

static void writeDesc (eventDesc p)

{

if (p == NULL)

{

NetworkOrder\_writeCard (file, ((unsigned int) (frameEvent)));

}

else

{

NetworkOrder\_writeCard (file, ((unsigned int) (p->etype)));

switch (p->etype)

{

case frameEvent:

break;

case circlesEvent:

writeCircles (p->cc);

break;

case circlePolygonEvent:

writeCirclePolygon (p->cp);

break;

case polygonPolygonEvent:

writePolygonPolygon (p->pp);

break;

case functionEvent:

writeFunction (p->fc);

break;

case springEvent:

writeSpring (p->sp);

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

}

}

/\*

writeEvent - writes the first event to the file.

\*/

static void writeEvent (eventQueue e)

{

MemStream\_Rewrite (file);

if (DebugTrace)

{

libc\_printf ((char \*) "time of next event in twoDsim is %g\\n", 37, e->time\_);

displayEvent (e);

}

if (DebugTrace)

{

libc\_printf ((char \*) "check time is %g\\n", 18, e->time\_);

}

NetworkOrder\_writeReal (file, e->time\_);

writeDesc (e->ePtr);

}

/\*

memDump -

\*/

static void memDump (void \* a, unsigned int len)

{

unsigned int i;

unsigned int j;

unsigned char \* p;

p = a;

j = 0;

for (i=0; i<=len; i++)

{

if ((j % 16) == 0)

{

libc\_printf ((char \*) "\\n%p %02x", 10, p, (unsigned int ) ((\*p)));

}

else

{

libc\_printf ((char \*) " %02x", 5, (unsigned int ) ((\*p)));

}

p += 1;

j += 1;

}

libc\_printf ((char \*) "\\n", 2);

}

/\*

checkOpened - checks to see of the MemStream file has been created and if not then

it is opened.

\*/

static void checkOpened (void)

{

ChanConsts\_OpenResults res;

if (! fileOpened)

{

fileOpened = TRUE;

MemStream\_OpenWrite (&file, ChanConsts\_write|ChanConsts\_raw, &res, &bufferStart, &bufferLength, &bufferUsed, TRUE);

if (res != ChanConsts\_opened)

{

libc\_printf ((char \*) "twoDsim.checkOpened: something went wrong when trying to open the memstream file (res = %d)\\n", 93, res);

libc\_exit (1);

}

}

}

/\*

Init -

\*/

static void Init (void)

{

trace = DebugTrace;

roots\_setTrace (trace);

maxId = 0;

objects = Indexing\_InitIndex (1);

framesPerSecond = DefaultFramesPerSecond;

replayPerSecond = 0.0;

simulatedGravity = 0.0;

eventQ = NULL;

freeEvents = NULL;

freeDesc = NULL;

currentTime = 0.0;

lastUpdateTime = 0.0;

lastDrawTime = 0.0;

drawCollisionFrame = FALSE;

drawPrediction = FALSE;

fileOpened = FALSE;

writeTimeDelay = TRUE;

haveSpringColour = FALSE;

haveCollisionColour = FALSE;

noOfCulledCollisions = 0;

startedRunning = FALSE;

framePolygons = FALSE;

freeBroadphase = NULL;

}

/\*

gravity - turn on gravity at: g m^2

\*/

void twoDsim\_gravity (double g)

{

simulatedGravity = g;

}

/\*

box - place a box in the world at (x0,y0), (x0+i,y0),

(x0+i, y0+j), (x0, y0+j).

\*/

unsigned int twoDsim\_box (double x0, double y0, double i, double j, deviceIf\_Colour colour)

{

return twoDsim\_poly4 (x0, y0, x0+i, y0, x0+i, y0+j, x0, y0+j, colour);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly3 - place a triangle in the world at:

(x0,y0), (x1,y1), (x2,y2)

\*/

unsigned int twoDsim\_poly3 (double x0, double y0, double x1, double y1, double x2, double y2, deviceIf\_Colour colour)

{

typedef struct \_T21\_a \_T21;

struct \_T21\_a { coord\_Coord array[2+1]; };

unsigned int id;

unsigned int i;

Object optr;

\_T21 co;

if (Debugging)

{

libc\_printf ((char \*) "begin poly3 (%g, %g, %g, %g, %g, %g)\\n", 38, x0, y0, x1, y1, x2, y2);

}

id = newObject ((ObjectType) polygonOb);

optr = Indexing\_GetIndice (objects, id);

co.array[0] = coord\_initCoord (x0, y0);

co.array[1] = coord\_initCoord (x1, y1);

co.array[2] = coord\_initCoord (x2, y2);

optr->p.nPoints = 3;

optr->p.cOfG = calculateCofG (optr->p.nPoints, (coord\_Coord \*) &co.array[0], 2);

optr->p.oldcOfG = optr->p.cOfG;

for (i=0; i<=optr->p.nPoints-1; i++)

{

optr->p.points.array[i] = polar\_coordToPolar (coord\_subCoord ((coord\_Coord) co.array[i], optr->p.cOfG));

}

optr->p.col = colour;

optr->p.mass = 0.0;

if (Debugging)

{

libc\_printf ((char \*) "end poly3\\n", 11);

twoDsim\_dumpWorld ();

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly4 - place a quadrangle in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3) the points must be in order,

either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly4 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, deviceIf\_Colour colour)

{

typedef struct \_T22\_a \_T22;

struct \_T22\_a { coord\_Coord array[3+1]; };

unsigned int id;

unsigned int i;

Object optr;

\_T22 co;

id = newObject ((ObjectType) polygonOb);

optr = Indexing\_GetIndice (objects, id);

co.array[0] = coord\_initCoord (x0, y0);

co.array[1] = coord\_initCoord (x1, y1);

co.array[2] = coord\_initCoord (x2, y2);

co.array[3] = coord\_initCoord (x3, y3);

optr->p.nPoints = 4;

optr->p.cOfG = calculateCofG (optr->p.nPoints, (coord\_Coord \*) &co.array[0], 3);

optr->p.oldcOfG = optr->p.cOfG;

for (i=0; i<=optr->p.nPoints-1; i++)

{

optr->p.points.array[i] = polar\_coordToPolar (coord\_subCoord ((coord\_Coord) co.array[i], optr->p.cOfG));

}

optr->p.col = colour;

optr->p.mass = 0.0;

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly5 - place a pentagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4)

the points must be in order, either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly5 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, deviceIf\_Colour colour)

{

typedef struct \_T23\_a \_T23;

struct \_T23\_a { coord\_Coord array[4+1]; };

unsigned int id;

unsigned int i;

Object optr;

\_T23 co;

id = newObject ((ObjectType) polygonOb);

optr = Indexing\_GetIndice (objects, id);

co.array[0] = coord\_initCoord (x0, y0);

co.array[1] = coord\_initCoord (x1, y1);

co.array[2] = coord\_initCoord (x2, y2);

co.array[3] = coord\_initCoord (x3, y3);

co.array[4] = coord\_initCoord (x4, y4);

optr->p.nPoints = 5;

optr->p.cOfG = calculateCofG (optr->p.nPoints, (coord\_Coord \*) &co.array[0], 4);

optr->p.oldcOfG = optr->p.cOfG;

for (i=0; i<=optr->p.nPoints-1; i++)

{

optr->p.points.array[i] = polar\_coordToPolar (coord\_subCoord ((coord\_Coord) co.array[i], optr->p.cOfG));

}

optr->p.col = colour;

optr->p.mass = 0.0;

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

poly6 - place a hexagon in the world at:

(x0,y0), (x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)

the points must be in order, either anticlockwise or clockwise.

\*/

unsigned int twoDsim\_poly6 (double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double x5, double y5, deviceIf\_Colour colour)

{

typedef struct \_T24\_a \_T24;

struct \_T24\_a { coord\_Coord array[5+1]; };

unsigned int id;

unsigned int i;

Object optr;

\_T24 co;

id = newObject ((ObjectType) polygonOb);

optr = Indexing\_GetIndice (objects, id);

co.array[0] = coord\_initCoord (x0, y0);

co.array[1] = coord\_initCoord (x1, y1);

co.array[2] = coord\_initCoord (x2, y2);

co.array[3] = coord\_initCoord (x3, y3);

co.array[4] = coord\_initCoord (x4, y4);

co.array[5] = coord\_initCoord (x5, y5);

optr->p.nPoints = 6;

optr->p.cOfG = calculateCofG (optr->p.nPoints, (coord\_Coord \*) &co.array[0], 5);

optr->p.oldcOfG = optr->p.cOfG;

for (i=0; i<=optr->p.nPoints-1; i++)

{

optr->p.points.array[i] = polar\_coordToPolar (coord\_subCoord ((coord\_Coord) co.array[i], optr->p.cOfG));

}

optr->p.col = colour;

optr->p.mass = 0.0;

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_xpos - returns the x coordinate of the center of gravity of object, id.

\*/

double twoDsim\_get\_xpos (unsigned int id)

{

double dt;

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

up ();

switch (optr->object)

{

case polygonOb:

return optr->p.cOfG.x;

break;

case circleOb:

return optr->c.pos.x;

break;

default:

libc\_printf ((char \*) "get\_xpos: only expecting polygon or circle\\n", 44);

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

break;

}

ReturnException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\*

get\_ypos - returns the first point, y, coordinate of object.

\*/

double twoDsim\_get\_ypos (unsigned int id)

{

double dt;

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

up ();

switch (optr->object)

{

case polygonOb:

return optr->p.cOfG.y;

break;

case circleOb:

return optr->c.pos.y;

break;

default:

libc\_printf ((char \*) "get\_ypos: only expecting polygon or circle\\n", 44);

M2RTS\_HALT (-1);

\_\_builtin\_unreachable ();

break;

}

ReturnException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\*

get\_xvel - returns the x velocity of object.

\*/

double twoDsim\_get\_xvel (unsigned int id)

{

Object optr;

if (trace)

{

libc\_printf ((char \*) "get\_xvel for object %d\\n", 24, id);

}

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

checkStationary (optr);

up ();

return optr->vx;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_yvel - returns the y velocity of object.

\*/

double twoDsim\_get\_yvel (unsigned int id)

{

Object optr;

if (trace)

{

libc\_printf ((char \*) "get\_yvel for object %d\\n", 24, id);

}

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

checkStationary (optr);

up ();

return optr->vy;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_xaccel - returns the x acceleration of object.

\*/

double twoDsim\_get\_xaccel (unsigned int id)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

up ();

return optr->ax;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

get\_yaccel - returns the y acceleration of object.

\*/

double twoDsim\_get\_yaccel (unsigned int id)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

up ();

return optr->ay;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

put\_xvel - assigns the x velocity of object.

\*/

void twoDsim\_put\_xvel (unsigned int id, double r)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

optr->vx = r;

checkStationary (optr);

up ();

}

/\*

put\_yvel - assigns the y velocity of object.

\*/

void twoDsim\_put\_yvel (unsigned int id, double r)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

optr->vy = r;

checkStationary (optr);

up ();

}

/\*

put\_xaccel - assigns the x acceleration of object.

\*/

void twoDsim\_put\_xaccel (unsigned int id, double r)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

optr->ax = r;

up ();

}

/\*

put\_yaccel - assigns the y acceleration of object.

\*/

void twoDsim\_put\_yaccel (unsigned int id, double r)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

optr->ay = r;

up ();

}

/\*

set\_rotate - assigns the y acceleration of object.

\*/

void twoDsim\_set\_rotate (unsigned int id, double angle)

{

Object optr;

printf("angle of rotation for object %d set to %g\n", id, angle);

down ();

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

optr->angleOrientation = angle;

up ();

}

/\*

apply\_impulse - apply an impulse to object, id,

along the vector [x, y] with magnitude, m.

Nothing happens if the object is fixed.

Currently only circles can have impulses

applied.

\*/

void twoDsim\_apply\_impulse (unsigned int id, double x, double y, double m)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

if (! optr->fixed)

{

switch (optr->object)

{

case circleOb:

apply\_impulse\_to\_circle (optr, x, y, m);

break;

default:

break;

}

}

up ();

}

/\*

moving\_towards - returns TRUE if object, id, is moving towards

a point x, y.

\*/

unsigned int twoDsim\_moving\_towards (unsigned int id, double x, double y)

{

Object optr;

unsigned int res;

down ();

optr = Indexing\_GetIndice (objects, id);

res = FALSE;

if (! optr->fixed)

{

switch (optr->object)

{

case circleOb:

res = circle\_moving\_towards (optr, x, y);

break;

default:

break;

}

}

up ();

return res;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

set\_colour - set the colour of object, id, to colour.

id must be a box or circle.

\*/

void twoDsim\_set\_colour (unsigned int id, deviceIf\_Colour colour)

{

Object optr;

// Get index of object

optr = Indexing\_GetIndice (objects, id);

// Switch based on kind of object and set appropriate varible to passed colour

// variable

switch (optr->object)

{

case polygonOb:

optr->p.col = colour;

break;

case circleOb:

optr->c.col = colour;

break;

// print error message if cannot be set

default:

libc\_printf ((char \*) "cannot set the colour of this object\\n", 38);

break;

}

}

/\*

set\_elasticity - set the elasticity of object, id, to elasticity.

id must be a box or circle.

\*/

void twoDsim\_set\_elasticity (unsigned int id, double elasticity)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

switch (optr->object)

{

case polygonOb:

case circleOb:

optr->elasticity = elasticity;

break;

default:

libc\_printf ((char \*) "cannot set the elasticity of this object\\n", 39);

break;

}

}

/\*

get\_elasticity - set the elasticity of object, id, to elasticity.

id must be a box or circle.

\*/

double twoDsim\_get\_elasticity (unsigned int id)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

switch (optr->object)

{

case polygonOb:

case circleOb:

return optr->elasticity;

break;

default:

libc\_printf ((char \*) "cannot get the elasticity of this object\\n", 39);

break;

}

ReturnException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\*

set\_gravity - set the gravity of object, id, to, g.

id must be a box or circle.

\*/

void twoDsim\_set\_gravity (unsigned int id, double g)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

switch (optr->object)

{

case polygonOb:

case circleOb:

optr->gravity = g;

break;

default:

libc\_printf ((char \*) "cannot set the gravity of this object\\n", 39);

break;

}

}

/\*

get\_gravity - return the gravity of object, id.

id must be a box or circle.

\*/

double twoDsim\_get\_gravity (unsigned int id)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

switch (optr->object)

{

case polygonOb:

case circleOb:

return optr->gravity;

break;

default:

libc\_printf ((char \*) "cannot get the gravity of this object\\n", 39);

break;

}

ReturnException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\*

get\_mass - returns the mass of object, id.

\*/

double twoDsim\_get\_mass (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

switch (idp->object)

{

case circleOb:

return idp->c.mass;

break;

case polygonOb:

return idp->p.mass;

break;

case springOb:

libc\_printf ((char \*) "should not be trying to use the mass of a spring\\n", 50);

return 0.0;

break;

default:

CaseException ("../git-pge/m2/twoDsim.def", 2, 1);

\_\_builtin\_unreachable ();

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

mass - specify the mass of an object and return the, id.

Only polygon (and box) and circle objects may have

a mass.

\*/

unsigned int twoDsim\_mass (unsigned int id, double m)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

switch (optr->object)

{

case polygonOb:

optr->p.mass = m;

break;

case circleOb:

optr->c.mass = m;

break;

default:

break;

}

calcInertia (optr);

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fix - fix the object to the world.

\*/

unsigned int twoDsim\_fix (unsigned int id)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

optr->fixed = TRUE;

up ();

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

unfix - unfix the object from the world.

\*/

unsigned int twoDsim\_unfix (unsigned int id)

{

Object optr;

/\* your code goes here... 3rd year and mcomp. \*/

down ();

optr = Indexing\_GetIndice (objects, id);

optr->fixed = FALSE;

optr->stationary = FALSE;

checkMicroInterpen ();

Assert (! optr->stationary, 687);

up ();

Assert (! optr->stationary, 689);

/\* end of 3rd year and mcomp. \*/

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

spring - join object, id1, and, id2, with a string of defined

by hooks constant, k, the spring is at rest if it has

length, l. If l < 0 then the game engine considers

the spring to naturally be at rest for the distance

between id1 and id2. The parameter, d, is used to

calculate the damping force.

\*/

unsigned int twoDsim\_spring (unsigned int id1, unsigned int id2, double k, double d, double l)

{

unsigned int id;

Object optr;

Object id1p;

Object id2p;

libc\_printf ((char \*) "spring\\n", 8);

libc\_printf ((char \*) "newObject\\n", 11);

id = newObject ((ObjectType) springOb);

libc\_printf ((char \*) "getIndice\\n", 11);

optr = Indexing\_GetIndice (objects, id);

if (l < 0.0)

{

libc\_printf ((char \*) "l < 0.0 lengthCoord\\n", 21);

l = coord\_lengthCoord (coord\_subCoord (getCofG (id1), getCofG (id2)));

}

libc\_printf ((char \*) "assign to optr\\n", 16);

optr->s.k = k;

optr->s.d = d;

optr->s.f1 = coord\_initCoord (0.0, 0.0);

optr->s.f2 = coord\_initCoord (0.0, 0.0);

optr->s.a1 = coord\_initCoord (0.0, 0.0);

optr->s.a2 = coord\_initCoord (0.0, 0.0);

optr->s.l0 = l;

optr->s.l1 = coord\_lengthCoord (coord\_subCoord (getCofG (id1), getCofG (id2)));

optr->s.id1 = id1;

optr->s.id2 = id2;

optr->s.hasCallBackLength = FALSE;

optr->s.draw = FALSE;

optr->s.drawEnd = FALSE;

optr->s.drawMid = FALSE;

/\* and stop the current position from being the next endPoint. \*/

anticipateSpring (currentTime, makeSpringDesc ((eventDesc) NULL, id, (history\_springPoint) history\_endPoint));

if (trace)

{

libc\_printf ((char \*) "in spring about to recalculate forces\\n", 39);

}

recalculateForceEnergy ();

if (trace)

{

libc\_printf ((char \*) "return from spring\\n", 20);

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

draw\_spring - draw spring, id, using colour, c, and a width, w.

\*/

void twoDsim\_draw\_spring (unsigned int id, unsigned int c, double w)

{

Object o;

o = Indexing\_GetIndice (objects, id);

if (isSpringObject (id))

{

o->s.drawColour = c;

o->s.draw = TRUE;

o->s.width = w;

}

else

{

libc\_printf ((char \*) "only spring objects can be modified by the draw primitive\\n", 59);

}

}

/\*

end\_spring - draw the spring using colour, c, when it reaches the end.

\*/

void twoDsim\_end\_spring (unsigned int id, unsigned int c)

{

Object o;

o = Indexing\_GetIndice (objects, id);

if (isSpringObject (id))

{

o->s.drawEnd = TRUE;

o->s.endColour = c;

}

else

{

libc\_printf ((char \*) "only spring objects can be modified by the end primitive\\n", 58);

}

}

/\*

mid\_spring - when the string reaches its rest point draw the objects

connected.

\*/

void twoDsim\_mid\_spring (unsigned int id, unsigned int c)

{

Object o;

o = Indexing\_GetIndice (objects, id);

if (isSpringObject (id))

{

o->s.drawMid = TRUE;

o->s.midColour = c;

}

else

{

libc\_printf ((char \*) "only spring objects can be modified by the mid primitive\\n", 58);

}

}

/\*

when\_spring - when the spring, id, reaches, length call, func.

\*/

void twoDsim\_when\_spring (unsigned int id, double length, unsigned int func)

{

Object o;

o = Indexing\_GetIndice (objects, id);

if (isSpringObject (id))

{

o->s.hasCallBackLength = TRUE;

o->s.cbl = length;

o->s.func = func;

}

else

{

libc\_printf ((char \*) "only spring objects can be modified by the when primitive\\n", 59);

}

}

/\*

circle - adds a circle to the world. Center

defined by: x0, y0 radius, r.

\*/

unsigned int twoDsim\_circle (double x0, double y0, double radius, deviceIf\_Colour colour)

{

unsigned int id;

Object optr;

id = newObject ((ObjectType) circleOb);

optr = Indexing\_GetIndice (objects, id);

optr->c.pos.x = x0;

optr->c.pos.y = y0;

optr->c.r = radius;

optr->c.mass = 0.0;

optr->c.col = colour;

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

velocity - give an object, id, a velocity, vx, vy.

\*/

unsigned int twoDsim\_velocity (unsigned int id, double vx, double vy)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

if (optr->fixed)

{

libc\_printf ((char \*) "object %d is fixed and therefore cannot be given a velocity\\n", 61, id);

}

else

{

optr->vx = vx;

optr->vy = vy;

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

accel - give an object, id, an acceleration, ax, ay.

\*/

unsigned int twoDsim\_accel (unsigned int id, double ax, double ay)

{

Object optr;

optr = Indexing\_GetIndice (objects, id);

if (optr->fixed)

{

libc\_printf ((char \*) "object %d is fixed and therefore cannot be given an acceleration\\n", 66, id);

}

else

{

optr->ax = ax;

optr->ay = ay;

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

rotate - gives object, id, an initial orientation.

\*/

unsigned int twoDsim\_rotate (unsigned int id, double angle)

{

Object optr;

if (! (roots\_nearZero (angle)))

{

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

if (optr->fixed)

{

libc\_printf ((char \*) "object %d is fixed and therefore cannot be given an angular velocity\\n", 70, id);

}

else

{

optr->angleOrientation = angle;

}

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

is\_visible - Gets boolean value for visibility.

\*/

unsigned int twoDsim\_is\_visible (unsigned int id)

{

return TRUE;

}

/\*

visibility - Sets boolean value for visibility

\*/

unsigned int twoDsim\_visibility (unsigned int id, unsigned int on)

{

Object optr = Indexing\_GetIndice(objects, id);

optr->visible = on;

return id;

}

/\*

rvel - gives object, id, an angular velocity, angle.

\*/

unsigned int twoDsim\_rvel (unsigned int id, double angle)

{

Object optr;

if (! (roots\_nearZero (angle)))

{

optr = Indexing\_GetIndice (objects, id);

checkDeleted (optr);

if (optr->fixed)

{

libc\_printf ((char \*) "object %d is fixed and therefore cannot be given an angular velocity\\n", 70, id);

}

else

{

optr->angularVelocity = angle;

}

}

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

fps - set frames per second.

\*/

void twoDsim\_fps (double f)

{

framesPerSecond = f;

}

/\*

replayRate - set frames per second during replay.

\*/

void twoDsim\_replayRate (double f)

{

replayPerSecond = f;

}

/\*

simulateFor - render for, t, seconds.

\*/

void twoDsim\_simulateFor (double t)

{

double s;

double dt;

/\*

gdbif.sleepSpin ;

\*/

startedRunning = TRUE;

s = 0.0;

if (twoDsim\_checkObjects ())

{

/\* avoid dangling else. \*/

if (s < t)

{

pumpQueue ();

if (trace)

{

printQueue ();

}

twoDsim\_dumpWorld ();

while (s < t)

{

dt = doNextEvent ();

s = s+dt;

}

updatePhysics (TRUE);

if (trace)

{

printQueue ();

}

}

}

else

{

libc\_printf ((char \*) "the game engine cannot be run as you have a moving object without a mass\\n", 74);

}

}

/\*

getTime - returns the current time in the simulation.

\*/

double twoDsim\_getTime (void)

{

return currentTime;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

skipTime - attempts to skip, t, seconds. It returns the amount

of time actually skipped. This function will not skip

past the next event.

\*/

double twoDsim\_skipTime (double t)

{

double dt;

if (Debugging)

{

libc\_printf ((char \*) "skipTime %f\\n", 13, t);

dumpTime ();

}

pumpQueue ();

if (eventQ == NULL)

{

libc\_printf ((char \*) " no events in the queue\\n", 26);

dumpTime ();

libc\_printf ((char \*) "finishing skipTime\\n", 20);

return 0.0;

}

else

{

if (t > eventQ->time\_)

{

dt = eventQ->time\_;

if (! (roots\_nearZero (dt)))

{

currentTime = currentTime+dt;

}

eventQ->time\_ = 0.0;

}

else

{

dt = eventQ->time\_-t;

if (roots\_nearZero (dt))

{

currentTime = currentTime+eventQ->time\_;

eventQ->time\_ = 0.0;

}

else

{

currentTime = currentTime+t;

eventQ->time\_ = eventQ->time\_-t;

}

}

if (Debugging)

{

dumpTime ();

libc\_printf ((char \*) "finishing skipTime\\n", 20);

}

return dt;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

timeUntil - returns the relative time from now until the next event.

\*/

double twoDsim\_timeUntil (void)

{

pumpQueue ();

if (eventQ == NULL)

{

return 0.0;

}

else

{

return eventQ->time\_;

}

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

processEvent - skips any outstanding time and processes the next event.

Time is adjusted to the time of the next event.

\*/

void twoDsim\_processEvent (void)

{

double dt;

/\* gdbif.sleepSpin ; \*/

if (Debugging)

{

libc\_printf ((char \*) "processEvent before pumpQueue\\n", 31);

}

pumpQueue ();

if (DebugTrace)

{

libc\_printf ((char \*) "before doNextEvent\\n", 20);

printQueue ();

}

dt = doNextEvent ();

if (DebugTrace)

{

libc\_printf ((char \*) "finished doNextEvent\\n", 22);

}

}

/\*

isCollision - returns TRUE if the next event is a collision event.

\*/

unsigned int twoDsim\_isCollision (void)

{

if (Debugging)

{

libc\_printf ((char \*) "isCollision before pumpQueue\\n", 30);

}

pumpQueue ();

return isEvent ((eventKind) collisionKind);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isFrame - returns TRUE if the next event is a frame event.

\*/

unsigned int twoDsim\_isFrame (void)

{

if (Debugging)

{

libc\_printf ((char \*) "isFrame before pumpQueue\\n", 26);

}

pumpQueue ();

return isEvent ((eventKind) frameKind);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isFunction - returns TRUE if the next event is a function event.

\*/

unsigned int twoDsim\_isFunction (void)

{

if (Debugging)

{

libc\_printf ((char \*) "isFunction before pumpQueue\\n", 29);

}

pumpQueue ();

return isEvent ((eventKind) functionKind);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isSpring - returns TRUE if the next event is a spring event.

\*/

unsigned int twoDsim\_isSpring (void)

{

if (Debugging)

{

libc\_printf ((char \*) "isFunction before pumpQueue\\n", 29);

}

pumpQueue ();

return isEvent ((eventKind) springKind);

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

isFixed - returns TRUE if object, id, is fixed.

\*/

unsigned int twoDsim\_isFixed (unsigned int id)

{

Object idp;

idp = Indexing\_GetIndice (objects, id);

return idp->fixed;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

createFunctionEvent - creates a function event at time, t,

in the future.

\*/

void twoDsim\_createFunctionEvent (double t, unsigned int id, unsigned int param)

{

eventQueue e;

eventDesc edesc;

pumpQueue ();

if (Debugging)

{

libc\_printf ((char \*) "function event %d will occur in %g simulated seconds\\n", 54, id, t);

}

edesc = newDesc ();

edesc->etype = functionEvent;

edesc->fc.id = id;

edesc->fc.param = param;

e = newEvent ();

e->kind = functionKind;

e->time\_ = t;

e->p.proc = doFunctionEvent;

e->ePtr = edesc;

e->next = NULL;

if (Debugging)

{

libc\_printf ((char \*) "queue before function event\\n", 29);

printQueue ();

}

addRelative (e);

if (Debugging)

{

libc\_printf ((char \*) "queue after function event\\n", 28);

printQueue ();

}

}

/\*

rm - delete this object from the simulated world.

The same id is returned.

\*/

unsigned int twoDsim\_rm (unsigned int id)

{

Object optr;

down ();

optr = Indexing\_GetIndice (objects, id);

if (DebugTrace)

{

libc\_printf ((char \*) "rm %d\\n", 7, id);

libc\_printf ((char \*) "here is the world before rm\\n", 29);

twoDsim\_dumpWorld ();

}

optr->deleted = TRUE;

if (DebugTrace)

{

libc\_printf ((char \*) "rm complete, here is the world after rm\\n", 41);

twoDsim\_dumpWorld ();

}

resetStationary ();

up ();

return id;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

/\*

getEventBuffer - collects the event buffer limits in the following parameters.

\*/

void twoDsim\_getEventBuffer (void \* \*start, unsigned int \*length, unsigned int \*used)

{

double \* f;

recordEvent ();

(\*start) = bufferStart;

(\*length) = bufferLength;

(\*used) = bufferUsed;

f = bufferStart;

if (DebugTrace)

{

libc\_printf ((char \*) "event buffer ptr = 0x%p, length = %d, used = %d\\n", 49, (\*start), (\*length), (\*used));

libc\_printf ((char \*) "ptr to real is %g\\n", 19, (\*f));

memDump ((\*start), 8);

}

}

/\*

buildFrame - populate the frame buffer contents with the world at the current time.

\*/

void twoDsim\_buildFrame (void)

{

deviceIf\_frameNote ();

drawFrame ((eventQueue) NULL);

deviceIf\_flipBuffer ();

}

/\*

emptyCbuffer - empty the colour buffer.

\*/

void twoDsim\_emptyCbuffer (void)

{

deviceIf\_emptyCbuffer ();

}

/\*

emptyFbuffer - empty the frame buffer.

\*/

void twoDsim\_emptyFbuffer (void)

{

deviceIf\_emptyFbuffer ();

}

/\*

useTimeDelay - should the frame buffer include the time delay command?

\*/

void twoDsim\_useTimeDelay (unsigned int on)

{

writeTimeDelay = on;

}

/\*

drawCollisionFrames - turn the drawing of collision frames on or off.

actual: determines whether an extra frame is generated

at the time of actual collision.

predict: draws a frame predicting the next collision.

It will show the points predicted to collide.

\*/

void twoDsim\_drawCollisionFrames (unsigned int actual, unsigned int predict)

{

drawCollisionFrame = actual;

drawPrediction = predict;

}

/\*

setCollisionColour - assigns, c, as the colour for objects colliding.

\*/

void twoDsim\_setCollisionColour (deviceIf\_Colour c)

{

collisionColour = c;

}

/\*

dumpWorld - dump a list of all objects and their characteristics.

\*/

void twoDsim\_dumpWorld (void)

{

unsigned int i;

unsigned int n;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

dumpObject ((Object) Indexing\_GetIndice (objects, i));

i += 1;

}

}

/\*

checkObjects - perform a check to make sure that all non fixed objects have a mass.

\*/

unsigned int twoDsim\_checkObjects (void)

{

unsigned int i;

unsigned int n;

Object optr;

unsigned int ok;

ok = TRUE;

n = Indexing\_HighIndice (objects);

i = 1;

while (i <= n)

{

optr = Indexing\_GetIndice (objects, i);

if (! optr->fixed && ! optr->deleted)

{

switch (optr->object)

{

case polygonOb:

if (roots\_nearZero (optr->p.mass))

{

libc\_printf ((char \*) "polygon %d is not fixed and does not have a mass\\n", 50, optr->id);

ok = FALSE;

}

break;

case circleOb:

if (roots\_nearZero (optr->c.mass))

{

libc\_printf ((char \*) "circle %d is not fixed and does not have a mass\\n", 49, optr->id);

ok = FALSE;

}

break;

default:

break;

}

}

i += 1;

}

return ok;

/\* static analysis guarentees a RETURN statement will be used before here. \*/

\_\_builtin\_unreachable ();

}

void \_M2\_twoDsim\_init (\_\_attribute\_\_((unused)) int argc, \_\_attribute\_\_((unused)) char \*argv[])

{

/\* gdbif.sleepSpin \*/

Init ();

}

void \_M2\_twoDsim\_finish (\_\_attribute\_\_((unused)) int argc, \_\_attribute\_\_((unused)) char \*argv[])

{

}