|  |
| --- |
| Nathan Chapman |
| S-kuru |
| Part I: The Defining, Planning & Designing of S-kuru |

|  |
| --- |
|  |

Table of Contents

[Defining the Problem and Its Solution 2](#_Toc289597739)

[Initial Ideas 2](#_Toc289597740)

[Investigate 3](#_Toc289597741)

[Needs of Users 3](#_Toc289597742)

[Objectives of the Software Solution 6](#_Toc289597743)

[Proposed Solutions 6](#_Toc289597744)

[Feasibility Study 7](#_Toc289597745)

[Recommendation 8](#_Toc289597746)

[Planning and Design 9](#_Toc289597747)

[Time Plan 9](#_Toc289597748)

[Interface Design 10](#_Toc289597749)

[Software Environment Boundaries 10](#_Toc289597750)

[Hardware Requirements 11](#_Toc289597751)

[Data Dictionary 11](#_Toc289597752)

[\_main\_.py 11](#_Toc289597753)

[playGame.py 12](#_Toc289597754)

[Data Validation 13](#_Toc289597755)

# Defining the Problem and Its Solution

## Initial Ideas

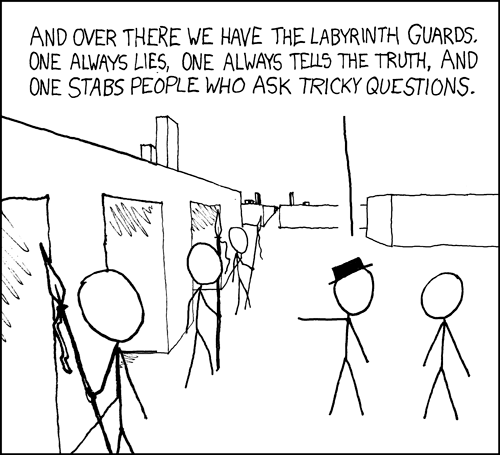
In starting this project I knew fairly early on the genre (or type) of game that I wanted to create: a puzzle game. This would satisfy the 'educational' criteria for the solution as a game requiring logic and specific thinking processes would be stimulating for all ages, including our target market, which is High School Students.

Figure : "Labyrinth Puzzle", from XKCD (http://xkcd.com/246/)

In the discussion amongst the class, we went through a variety of games that we could make. Many of these suggestions, however, did not appeal to me as they were merely remakes of existing games. Some of those mentioned were Minesweeper, a game bundled with Windows since 1989[[1]](#footnote-1); Tic-Tac-Toe, a well-known game played on paper and computers, and featured in the film War Games; or Solitaire, another game bundled with Windows.

What I was looking for was an original concept that would challenge the user into actually thinking about their next move. S-kuru was in a few parts inspired by World of Goo, a puzzle game concerning the building of towers using goo balls that stick together. Some of the aspects I liked about this game were the lack of a time limit. Many puzzle games are only difficult because they feature by an overly restrictive time limit which creates pressure on the user. This pressure makes the game in question far less enjoyable than one that is difficult purely because of its puzzles or an ungainly *core mechanic*.

Throughout this report I am likely to refer to the term *core mechanic* so I will define it now. A game's *core mechanic* is its major premise and way of working. For example, Minesweeper's *core mechanic* is the number in each cell that reveals the number of surrounding mines. The term comes from Dungeons & Dragons, where the *core mechanic* simply refers to its use of a d20 (a twenty-sided die) to determine success or failure of an action. Because of the importance of the *core mechanic* to any game, but particularly a puzzle game, I spent some time brainstorming it and getting it just right. Actually brainstorming on paper, here is the *core mechanic* that I finally came up with:

The aim of Sākuru is to get every ball on the screen to its exit. There are multiple coloured balls and exits; blue ball will only use the blue exit. Circles placed on the screen by the user can be travelled through by balls, and thereby get to their designated exit.

This is Sākuru's *core mechanic*, and I feel that it is simple enough to grasp fully after a single tutorial level or README file, yet it is complex enough to provide flexibility for the creation of difficult levels. The introduction of different coloured balls and circles was not made until fairly late in the process, but it adds much to the game that will prevent any repetition in levels. Another reason that this game appeals to me is because I would play this myself. The concept intrigues me and the name Sākuru would pique my curiosity (by the way, Sākuru is Japanese for "Circles". Using a Japanese word for the name of a project is commonplace within Ruby programming circles; however I thought it could be spread to Python without much trouble from the community).

## Investigate

### Needs of Users

I conducted a survey amongst both specifically chosen participants that were spread across several age groups (respondents were grouped into age brackets between 12 and 20 years old) and an open invitation to participate. Differentiating between these two sets of data, it becomes evident the differing requirements of a game for both High School students and adults. The survey received a total of 18 respondents over a three-week period.

For a copy of the survey questions and full results without interpretation, see [Appendix I].

The survey was in three sections as follows:

#### General Gaming Statistics

As the project specifications indicated that the software solution had to be a game, it was logical for the survey to ask how the potential users play games, and for how many hours each week. This would help to determine how the game should progress and the difficulty curve.

Question 1, which asked about the gaming habits of respondents, gave a clear answer to this (See Figure 2). From the results 71% of people play less than 6 hours per week (which I would classify as a 'casual' gamer), so the game has to have easy save/load points, to enable users to put down the game and come back at a later date/time.

Figure : Hours spent gaming per week

The next two questions were to gauge the possible audience for Sākuru: the first was to find popularity for various 'genres' of gaming, and the second asked specifically regarding puzzle games.

Diving the gaming world into genres is a difficult task, as it poses problems similar to literature, where the previously separate genres of Sci-Fi and Fantasy have merged now to such a degree[[2]](#footnote-2) where they are indistinguishable and have warranted the creation of an ‘umbrella genre’: Speculative Fiction[[3]](#footnote-3). In the same way, game genres vary; however for this survey I labelled games according to their gameplay style as opposed to their content. In this way, I came up with nine major categories that I felt were sufficient to categorise most games. This question was structured as a ‘matrix’, meaning that each option could be given a rating out of 5 according to how much the respondent plays or enjoys that genre of games.

From this matrix I have assigned each possible answer from 5 (“I live for these games “) to 1 (“I don’t play this type of game”) a score between 4 and 0. This creates their rankings as presented in the graph below. From the results (See Figure 3) it becomes plain that although the long-form RPG game is the most popular game (although only by two points, scoring 23 and beating Puzzle and FPS on 21). That Puzzle games ranked the second highest in this question is promising as it shows that the target audience for my Major Project will actually be interested in playing the game.

Figure : Ranked scores of popular game genres

The last question in this section asked specifically whether they liked and played puzzle games. This question also allowed for a long-response answer to justify or explain their response.

From the closed-answer part of this question (See Figure 4) it is very obvious that respondents enjoy this form of gameplay. The individual responses that accompany this question also tell a similar tale. Almost all of the open responses were positive along the lines of this one:

I like that feeling of accomplishment you get from solving a well-made puzzle. That and I like seeing the clever things people can do based on what are fairly simple mechanics. For example Braid would take something simple like mapping the passage of time to the player's position on the horizontal plane (you walk forwards, time goes forwards, you go backwards and time, well you get the point) and build all these elaborate puzzles around it that you really had to think to solve. So keep a minimal amount of mechanics and do something cool with them.

Figure : Do you play or enjoy puzzle games?

Most of the responses, in fact, all commented on how puzzle games are played differently to any other genre. For example, one respondent described first-person shooters as “HEY LOOK I SHOOT YOU NOW BANG BANG BANG”. The only negative open response that I received was this:

Requires too much thought...

This response was in direct contrast to the rest of the open answers. With both questions 2 and 3 showing such a positive reaction and appreciation of puzzle games, it was a logical step to further plan and to finally begin creating S-kuru. There is also emerging evidence amongst the scientific community that a well-developed puzzle game can help children’s (and teenagers’) brain development, including deep analytical problem-solving skills[[4]](#footnote-4).

#### Aspects of Effective Game Design

This section had several questions that asked the impressions potential users got from some Photoshop mockups of the user interface during the game. These questions were just for my reference, I didn’t plan to get any statistical data from them. Question 4, however, was important in working out what appealed most to users when playing a game.

Another matrix-style question, the ranked scores are as in Figure 5. From this we can see that the most important thing to consider when creating a game (and specifically a puzzle game) is the Gameplay (or ‘*core mechanic*’, as I’ve called it above). The visual style of the game came as a close second concern for users, and I have spent much time developing a coherent and intuitive visual style for the game that helps to convey the purpose of each element whilst retaining a comprehensible feel.

Figure : Importance of aspects of a game

This trend that is evident from this survey corresponds to real life and large-scale, triple-A titles also. Editor in Chief of Edge Magazine, Tony Mott had this to say about game design, which could also be interpreted as being true for any software solution[[5]](#footnote-5):

Ocarina of Time is nearly ten years old, but its position at the top slot in Edge’s 100 Best Videogames shows that great game design does not age. In visual terms, it obviously cannot compete with today’s Xbox 360 and PS3 productions, but, as with many classics, its appeal is about so much more than its appearance.

### Objectives of the Software Solution

From the results given during the survey I am able to formulate some measureable objectives that I was S-kuru to achieve. I do not feel that these are easily obtainable (otherwise all software solutions would implement them), however it is not outside my reach by any means.

1. Simplicity of Use – the game must flow and play smoothly, and the controls must be intuitive. Not only in which button does what, but in (for example) how fast a circle grows as the user holds their mouse down. More measureable aspects of this include:
   1. Simple and feedback about progress of a level. Clear feedback about progress through the game.
   2. Anything that is ‘clickable’ throughout the game should be labeled or designed in such a way as to make it visually so.
2. Complexity of Thought – the game should provoke thought within the program’s use and force the user to consider each move and its implications within the game.
   1. Puzzles (i.e. individual levels) should provide a differing experience as they progress. They should not be repetitive.
   2. There should be multiple solutions to puzzles, even if some of these were not originally considered by the level designer.
3. Consistency whilst constantly evolving – the game should shift and change constantly so as to be become stagnant and dull. Throughout this change, however, the gameplay *core mechanic* should remain the same, as will level design. This allows for a consistent user experience throughout a varying game.

## Proposed Solutions

There are several possible alternatives that would still fulfill the objectives that are outlined above, as well as address the opinions of the potential users in the survey.

Any computerized remake of an existing puzzle game, such as Sudoku, Scrabble or a Cross-Word would achieve both of these objectives. The appreciation of puzzle games that is evident in the survey, however a game created from the single most popular genre, a Role-Playing Game, would present the most appropriate alternative solution.

To still fulfill many of the objectives above, the game would have to be structured in a similar way to Legend of Zelda: Ocarina of Time. In this game, a mix of puzzle and adventure all tied together with a compelling and strong storyline create a fantastic game that is among the best of all time.[[6]](#footnote-6) To create a game on such a scale as Ocarina of Time would be a grand undertaking; however it would accomplish everything above, in addition to satisfying the desires of the potential users from the survey.

## Feasibility Study

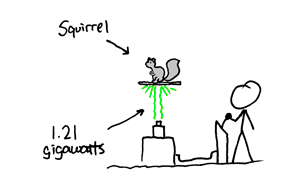
To help evaluate these two proposed solutions, each solution will be assigned a score out of 10 for each dot point in addition to the written appraisal.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Point to Address | S-kuru | | RPG | |
| Is the solution technically feasible? | Once the physics ‘engine’ is coded and completed, the rest of the solution is easily put in place. Level design will depend more upon skill and ideas of the creator, rather than the program’s technical aspects. | 8.5 | Featuring (by definition) a large variety of possible interactions with the environment, the sheer amount of events to cater for would make the task a major undertaking. An engaging story, similar to the level design in S-kuru would depend on the skill of the writer. | 5 |
| Is it operationally feasible? | As a 2D puzzle game, the hardware requirements for the game are little. Preliminary tests with unoptimised code have showed that the physics engine can run at 60fps with 3 circles and 25 balls without lag – far outside the operational requirements within the game. | 9.5 | Depending on whether the game would be 2D or 3D based, the hardware requirements would vary. Having said this, even low-end modern computers can run semi-complex 3D games (as most new computers come with on-board graphics), so this would be of little issue. | 9 |
| Is the project financially feasible? | For a small development studio, a game on this scale could be pushed out within a few weeks, and depending on the publicity for the game would make some profit. In this way, the game could be financially feasible. For a student to develop, the program will be an exhaustive process over several months, as the engine and level framework that the puzzles work from would take time to develop. | 6 | This game, because of the epic scale, would require a large development studio (which is expensive), though it would be practically guaranteed of commercial success. | 8 |
| Can it be completed in the required time frame? | Development will take several months, however once the physics engine and the game framework is created, levels can be created both by the Programmers and by dedicated level designers, shortening development time. | 8 | Due to the nature of the Structured Development Cycle that is required when working with a large team of developers, the time frame for this solution would be immense[[7]](#footnote-7). | 4 |
| Are there any social and ethical issues to address? | Possible issues would include:   * Cheating will be easy, as levels are stored as plain-text files * Viruses could use vulnerabilities in programming language (Python) or used modules (PyGame, YAML) to infiltrate system as these are open-source * Incorrect use of licenses for these software packages | 6.5 | Possible issues would include:   * Pricing of such a game * Gender bias * License Agreements * Workplace environment for development team (ergonomics) * Piracy * Virus distribution | 5 |

## Recommendation

|  |  |
| --- | --- |
| Solution | Final Score |
| S-kuru | 38.5 |
| RPG | 31 |

Having considered the two possible solutions using the table above, it is clear which would be a more feasible and effective solution to the problem.

Out of a total possible score of 50, S-kuru scored higher in every category bar financial feasibility. It scored lower in this section because the produced product would not be sold, but distributed free. Having no monetary return (as opposed to a large game like the proposed RPG), it would prove to be less financially feasible for the amount of development time that the game will take.

Even then, the RPG’s higher score for that category assumes that it is being sold and will make a profit. In every other category, the RPG scores lower due to its inherit complexity that is required to make the game engaging.

S-kuru is a far simpler, more easily accomplished solution that also fulfills the objectives from above whilst at the same time looking sophisticated and being a marketable, a cohesive solution to the problem.

Figure : This is not a good recommendation. From XKCD (http://blog.xkcd.com/2008/02/15/the-laser-elevator/)

# Planning and Design

## Time Plan

## Interface Design

E:\S-kuru\01_Design\Sprites\Blue-Ball.jpgE:\S-kuru\01_Design\Sprites\Grey-Ball.jpgFrom the very beginning of active development, I have worked from an interface mockup (See Figure 7) of what I wanted to achieve as the final product that would become S-kuru. Many of the aspects of this original design constitute good design techniques, although some have been purposely neglected for effect.

One aspect that has been purposefully neglected was colour: the game is wholly black and white. This is to add a ‘new’ type of puzzle to the game in later stages, where the balls become coloured, and will only travel through circles that are made up of their colour (See Figure 8). This adds a new puzzle element to the game, making it a more complex and engaging experience for the user. Although the low-contrast design of the game may have some effect on colourblind or vision-impaired users, this could easily be rectified by implementing a high-contrast mode that gives the game a new colour scheme.

Figure : Differing types of balls

Figure : Original Mockup of S-kuru Interface

## Software Environment Boundaries

Any software solution will have restrictions from outside the system that affect the program. For S-kuru, these restrictions are as follows:

* DET Software Restrictions – with the rollout of the D.E.R. Laptops to all year 9 students, it would seem that a primary user base for an engaging, thought-provoking puzzle game would have been found easily. However, due to DET restrictions on software use, external applications cannot be installed or run on these machines. This limits the potential user base; however it could be circumnavigated by distributing the .py source code, which could be run using the pre-installed Python Interpreter.
* Library Modules – being another developer’s intellectual property, any modules that have been used in the development of S-kuru (such as PyGame or YAML) would have their own license agreements which need to be adhered to. It is likely that these specify the license that S-kuru must be distributed under, or whether it can be commercially sold.
* Client Requirements – the game needs to fulfill the assignment specifications for the Software Design & Development course. In a real-world setting, this would mean tailoring the solution to the client’s needs and their current system.

## Hardware Requirements

The requirements to run S-kuru are very low, due to its use of simple 2D graphics and an optimized physics engine for collisions. As such, the requirements below are not to be considered the absolute minimum that the game will run on; instead, they are a broad guide.

|  |  |  |
| --- | --- | --- |
| Type | Requirements | Justification |
| Operating System | Windows, Linux/Unix and Mac OSX. | Any operating system that supports Python. |
| Processor | 1.6 GHz Single Core | Minimum required for Windows 7[[8]](#footnote-8), plus some extra processing for game itself. |
| RAM | 1GB | Can run with 20 colliding balls with no loss of framerate from 60 fps. |
| HDD | 50MB | This is the total size of installed Python. The distributable will be far smaller. |
| Video Card | Not required | Game has no 3D capabilities and hardware acceleration has not been implemented as it is unnecessary for this solution |
| Screen | 1024 x 768; 32-bit colour | Game runs at 800 x 600, 32-bit colour |
| Other | Full keyboard, mouse | Used to control game; mainly because S-kuru is not compatible with touch-screen devices |

## Data Dictionary

### \_main\_.py

This module is the parent module for the whole program. It runs the main menu, and from there the program ‘drops down’ into sub-modules for levels, editor etc.

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose |
| height / width | Integer | Specify in pixels the height/width of the game window |
| screen | Object based on PyGame surface() class. | To provide a surface to display all objects etc. |
| FPSClock | Object based on PyGame Clock() class. | Used with tick() to limit frame rate in-game |
| running | Boolean | Used as a flag to quit the game |
| buttonTextList | List of strings | Holds text of buttons |
| buttons | List | List of Button() objects |
| newLogo | Instance of Logo() class | Used to display bitmap in game |
| u | One object from buttons list | Used in for loop to run function on each instance of the Button() class. |
| event | A PyGame event from the stack | Used in for loop to test for events such as pygame.quit() etc. |
| mousePos | Tuple | Created from pygame.mouse.get\_pos(), used to check mouse position |

### playGame.py

This module is called and takes over from the main loop of S-kuru when the ‘Play’ button is pressed in the main menu. It handles loading the next level, displaying the objects and running the level.

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose |
| balls / circles / exits | Lists of objects | Three separate lists used to hold instances of respective classes |
| gravity | Tuple | Gravity constant used for physics calculations |
| drag | Float | Drag constant used for physics calculations |
| elasticity | Float | Elasticity constant used for physics calculations |
| hit | Boolean | Used as a flag during collision calculations |
| closestDist | Integer | Value to test for closest circle to colliding ball |
| dy / dx | Integer | Used to test total X / Y distance between two objects |
| distance | Float | The hypotenuse of a triangle formed from dy, dx and distance; exact distance between the two objects |
| levelClock | Instance of PyGame Clock() class | Used to limit frame rate |
| loadText / loadCircles / loadBalls / loadExits | Lists of strings | Used to parse level data from YAML into game |
| c / b / e | One object from balls / circles / exits list | Used in for loop to run function on each instance of the respective class |

Each of these variables are then passed to various sub-modules, where they are passed to a function which then can display the variable (for example, the newLogo variable) or do a calculation to be passed back into the parent module (again, the dy / dx and distance of balls passed to the module\_physicsEngine.py module to calculate the angle of ‘bounce’).

[[SUBMODULES STILL NEED TO BE COMPLETED!!]]

## Data Validation

Any input from the user has to be checked for validity, as user errors are the most common cause of program failure. In S-kuru, the most common method of user input is the mouse and keyboard.

When playing the game, the position of the mouse within the game window is calculated from a built-in function within the PyGame Mouse module, pygame.mouse.get\_pos(). This allows for a simple method of checking if the position is within the width and height of the window to validate mouse input. For mouse buttons, the following code shows how the position of a mouse ‘click’ is validated and checked against the position of buttons in the game:

if event.type == pygame.MOUSEBUTTONDOWN:

mousePos = pygame.mouse.get\_pos()

for item in buttons:

X = item.getXPos()

Y = item.getYPos()

if X[0] < mousePos[0] < X[1] and Y[0] < mousePos[1] < Y [1]:

item.action(screen)

As we can see from the above section of code, the position of the mouse is gotten, and then compared to the position of each button in a list. If the mouse position is within a button, the button’s function is executed.

When the user is inputting data using the keyboard, since Unicode characters are not compatible with Python by default they will not be parsed at all – this allows for one less step in data validation, as only on-keyboard characters have to be dealt with when the user is typing text (for example, when typing their name for a high-score table etc.).

When actually playing the game, various keys are used to change the colour mode of the circles. PyGame’s event.type() function is structured in such a way as that for a key to have an action, it must be specified in the program, similarly to this:

if event.key == pygame.K\_DOWN

Using this method for handling keyboard presses is also useful for keyboard validation as any key that is not specified to have an action is simply ignored when depressed, as a form of self-validation by PyGame (and S-kuru).

## Structure Diagram

\_main\_

Module\_text

Module\_graphics

playGame

editLevel

buttonTextList

buttons

newLogo

Balls, circles

pauseMenu

Module\_fileHandling

Module\_text

Module\_physicsEngine

levelName

levelData

text

Text (Instances of Class)

b, c

levelName

bounceFlag

Note: these are the same for both modules’ use of *pauseMenu*

levelName

running

running

running

Module\_fileHandling

saveFlag

levelName,  
levelData

## Data Flow Diagrams

### Overall System Flowchart (Context Diagram)

Keyboard

Mouse

\_main\_.py

Monitor

File Handling

HDD Storage

Play Game

Edit Level

Main Menu - \_main\_.py (DFD)

User

MOUSE  
MOVE

MOUSE  
CLICK

OVER BUTTON TRUE

*‘running’* BOOLEAN FLAG

*‘running’* BOOLEAN FLAG

*‘running’* BOOLEAN FLAG

### Play Game (DFD)

Please note that due to an unavailability of the correct three-sided rectangle symbol for “A file or data storage”, a parallelogram will be used instead.

User

fileHandling

LEVEL DATA

MOUSE / KEYBOARD DATA

PYGAME.EVENT

ARRAYS OF CLASS INSTANCES

INDIVIDUAL DATA ARRAYS

INDIVIDUAL DATA ARRAYS

INDIVIDUAL DATA ARRAYS

KEYBOARD / MOUSE DATA

User

fileHandling

LEVEL DATA

INDIVIDUAL DATA ARRAYS

INDIVIDUAL DATA ARRAYS

PYGAME.EVENT

INSTANCES OF CLASSES IN ARRAYS

INSTANCES OF CLASSES IN ARRAYS

FINISHED LEVEL

### Level Editor – levelEditor.py (DFD)

## Algorithms

### Overall Program

The following flowchart describes the thought process behind the overall structure of how S-kuru works. Each module has its own game ‘loop’, and when a sub-module is initiated (such as moving from the main menu to playing the game, and then further into the pause menu) the program ‘drops down’ into a new loop.

BEGIN

Button = Quit

END

Button = Play

PlayGame

Button  
= Clicked

editLevel

T

T

T

F

F

F

### Physics Module

The most interesting part of S-kuru is the custom-built physics engine that was written for both external and internal circular collisions. External collisions are used when two balls collide, and the internal collisions are for when a ball is colliding within a circle. Other functions within the Physics Engine include things like adding vectors and calculating the elastic bounce required.

To avoid repetition of code in the program, many of the test statements were replaced with functions from the Physics Engine sub-module several revisions ago. This makes many of these algorithms interrelate. Also remember that both circles and balls are lists of instances of the respective class (for example, balls is a list of the Ball() class).

#### External Collisions – collideBalls()

BEGIN SUBPROGRAM collideBalls with b1 and b2  
 IF collideTest with b1 and b2 = True THEN  
 ballBounce with b1 and b2  
 ENDIF  
END SUBPROGRAM

#### Internal Collisions – collideCircle()

BEGIN SUBPROGRAM collideCircle with ball and circles  
 Set hit to False  
 Set closestDist to 0  
 FOR c = 0 to (circles.len-1)  
 dx = circles[c].x – ball.x  
 dy = circles[c].y – ball.y  
 calculateHypotenuse with dx and dy

*# In python, calculating the hypotenuse given two sides is handled by the Math() built-in module, so there is no algorithm for this function*  
 IF hypotenuse <= (c.size – ball.size) THEN  
 *# This would mean that the ball is inside any circle*  
 Set hit to False  
 ELSE  
 *# The ball is outside of this particular circle*  
 IF closestDist < c.size – (hypotenuse – ball.size) THEN  
 *# If this is the closest circle the ball is to* Set currentCircle to circles[c]  
 Set hit to True  
 Set closestDist to (c.size – (hypotenuse – ball.size))  
 ENDIF  
 ENDIF  
 ENDFOR  
 IF hit = True THEN  
 circleBounce with currentCircle and Ball  
 ENDIF  
END SUBPROGRAM

#### Adding Vectors – addVector()

BEGIN SUBPROGRAM addVector with angle1, length1 and angle2, length2  
 Set sin1 to calculateSin with angle1  
 Set sin2 to calculateSin with angle2  
 Set cos1 to calculateCos with angle1  
 Set cos2 to calculateCos with angle2  
 x = sin1 \* length1 + sin2 \* length2  
 y = cos1 \* length1 + cos2 \* length2  
 Set length to CalculateHypotenuse with x and y  
 Set atan to CalcuateAtan with y and x  
 # Similarly to before, this function is handled by the Python Math.atan2(y,x) function, and as such there is no algorithm for it  
 angle = 0.5 \* 3.141[[9]](#footnote-9) - atan  
 RETURN angle and length  
ENDSUBPROGRAM

#### Collision Testing – collideTest()

#### BEGIN SUBPROGRAM collideTest with A and B x = A.x – B.x y = A.y – B.y calculateHypotenuse with x and y IF hypotenuse <= (A.size + B.size) THEN RETURN True ELSE RETURN False ENDIF END SUBPROGRAM

#### Elastic Reaction during Collision – bounceBall() BEGIN SUBPROGRAM ballBounce

#### def ballBounce(A, B):

#### """Calculate the correct changes to each ball after a collision"""

#### dx = A.x - B.x

#### dy = A.y - B.y

#### tangent = math.atan2(dy, dx) # Find the tangent of the point

#### angle = 0.5 \* math.pi + tangent # We use this later on

#### A.angle = 2\*tangent - A.angle # Alter angles

#### B.angle = 2\*tangent - B.angle

#### if A.speed != 0 and B.speed != 0:

#### (A.speed, B.speed) = (B.speed, A.speed) # Swap speeds

#### A.speed \*= elasticity # Reduce speed due to elasticity

#### B.speed \*= elasticity

#### A.x += math.sin(angle) # Move particles away from each other

#### A.y -= math.cos(angle)

#### B.x -= math.sin(angle)

#### B.y += math.cos(angle)

1. http://www.gamesetwatch.com/2007/02/column\_beyond\_tetris\_minesweep.php [↑](#footnote-ref-1)
2. http://www.depauw.edu/sfs/review\_essays/elkins22.htm [↑](#footnote-ref-2)
3. http://www.writepop.com/writing/naming-a-genre [↑](#footnote-ref-3)
4. http://abcnews.go.com/WNT/Health/story?id=814080 [↑](#footnote-ref-4)
5. http://www.wired.com/gamelife/2007/07/edge-names-ocar/ [↑](#footnote-ref-5)
6. As said by Rhys Powell; also http://www.next-gen.biz/features/the-100-best-games-to-play-today [↑](#footnote-ref-6)
7. The game that this solution got its inspiration from, The Legend of Zelda: Ocarina of Time, spend over 3 years in active development before the Nintendo Entertainment Analysis and Development deemed it ready for release. [↑](#footnote-ref-7)
8. http://windows.microsoft.com/en-AU/windows7/products/system-requirements [↑](#footnote-ref-8)
9. In this actual program a more precise version of Pi is calculated using Math.Pi() [↑](#footnote-ref-9)