# System Maintenance

Contents

[System Maintenance 1](#_Toc414554066)

[Introduction 2](#_Toc414554067)

[Embedded Resource File Explained 2](#_Toc414554068)

[User Interface 2](#_Toc414554069)

[Introduction to User Interface 2](#_Toc414554070)

[Main Menu Form 3](#_Toc414554071)

[Create your own Model Form 6](#_Toc414554072)

[Practice Form 9](#_Toc414554073)

[Description of System Settings 12](#_Toc414554074)

[Functionality 13](#_Toc414554075)

[Introduction to Functionality 13](#_Toc414554076)

[BallUnit/TBall Object Overview 13](#_Toc414554077)

[MainMenuUnit 15](#_Toc414554078)

[OwnModelUnit 17](#_Toc414554079)

[PracticeUnit 44](#_Toc414554080)

[Code Appendix 69](#_Toc414554081)

[BallUnit 69](#_Toc414554082)

[MainMenuUnit 71](#_Toc414554083)

[OwnModelUnit 73](#_Toc414554084)

[PracticeUnit 94](#_Toc414554085)

## Introduction

Throughout this guide, if you would like to reference the raw code, see the Code Appendix at the end of the guide.

This guide will covers how the Interactive ‘Pulley Systems on an Incline’ program has been created so that it can be maintained. Any maintenance should be done using the program source code file within a Delphi-Firemonkey compiler. An understanding of the Edexcel M1 Mathematics syllabus and Delphi-Firemonkey is recommended. More information on how the functionality is presented to the user not found in this guide can be found in the User Manual.

The programming language used was Delphi-Firemonkey. This choice was made because the language allows different traversable forms to be created with ease while also being event-driven. It also allows for a compiled executable to be created with an embedded resource file.

The program is split into two main parts. In the first part (Part A or ‘Create your own Model’), the user is able to create their own M1 ‘pulley systems on an incline’ model, while the second part questions the user on a randomly generated model (Part B or ‘Practice’).

To this end, the program is split into 3 forms. One as a main menu, one for Part A, and one for Part B. These Forms are called MainMenuForm, OwnModelForm and PracticeForm.

The program uses 4 units. One for each of the three forms, and one for an object (TBall) which can be called by any of the other units. These Units are called MainMenuUnit, OwnModelUnit, PracticeUnit and BallUnit.

## Embedded Resource File Explained

Delphi-Firemonkey allows for a resource file to be embedded within the compiled application. In this program, two files are stored within this resource file. Firstly, ‘BallImage’. This is an image file used to display the balls used in the problems on screen. Secondly, ‘Questions’. This is a text file storing all the potential questions to be used in the application.

## User Interface

### Introduction to User Interface

Over the next few pages are two screenshots and a table for each of the three forms during program execution. One screenshot is ‘clean’, and as the user would see. The other has all elements of the form labelled with their coded names. The table defines each element based on what class it is.

Note: In M1 ‘Plank’ and ‘Incline’ are interchangeable

### Main Menu Form

#### Main Menu Clean



#### Main Menu Annotated



GotoOwnModelButton

GoToPracticeButton

CloseProgramButton

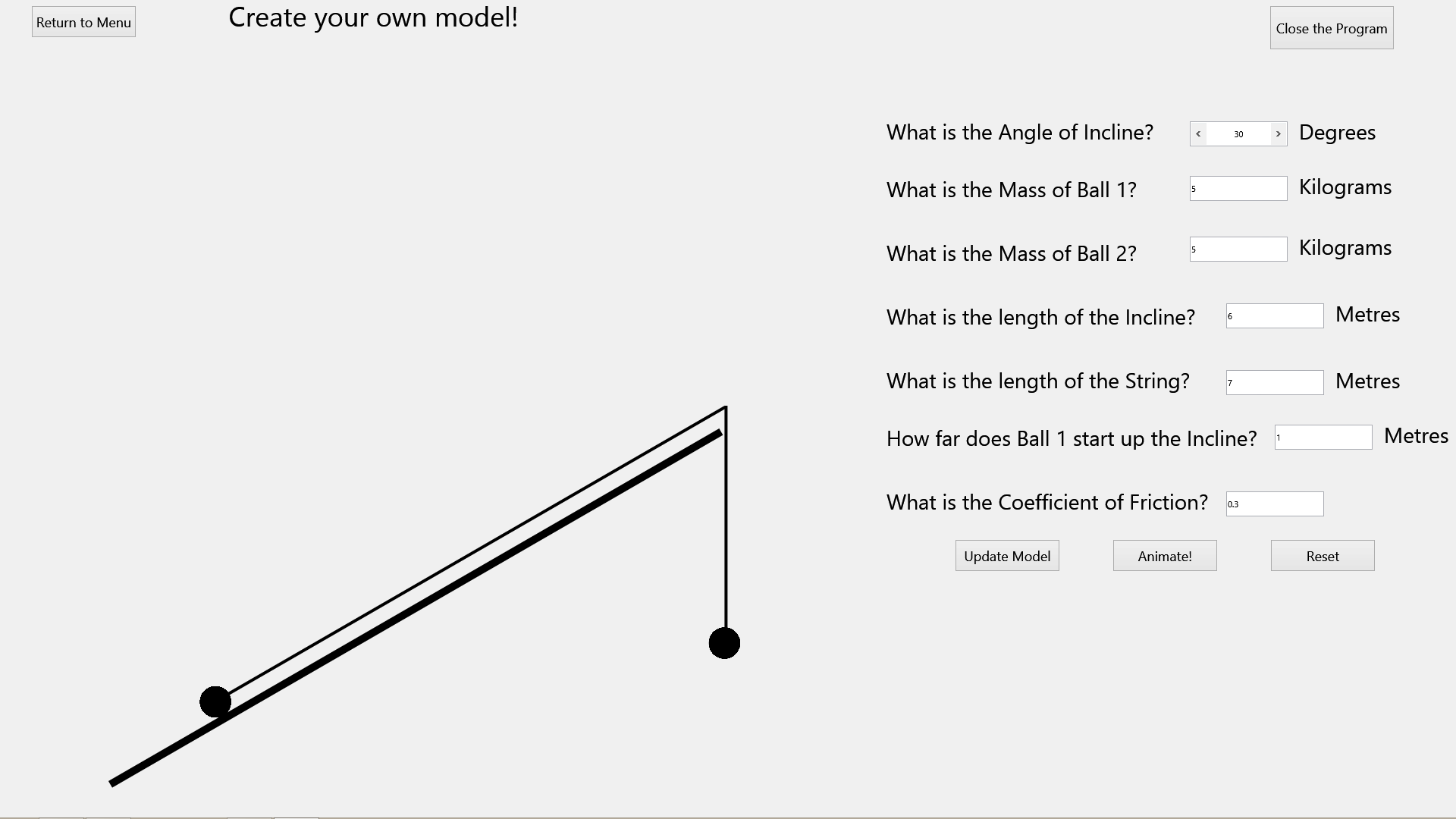
WelcomeMessage

#### Main Menu Table

|  |  |
| --- | --- |
| **Element** | **Type** |
| WelcomeMessage | TLabel |
| GoToOwnModelButton | TButton |
| GoToPracticeButton | TButton |
| CloseProgramButton | TButton |

### Create your own Model Form

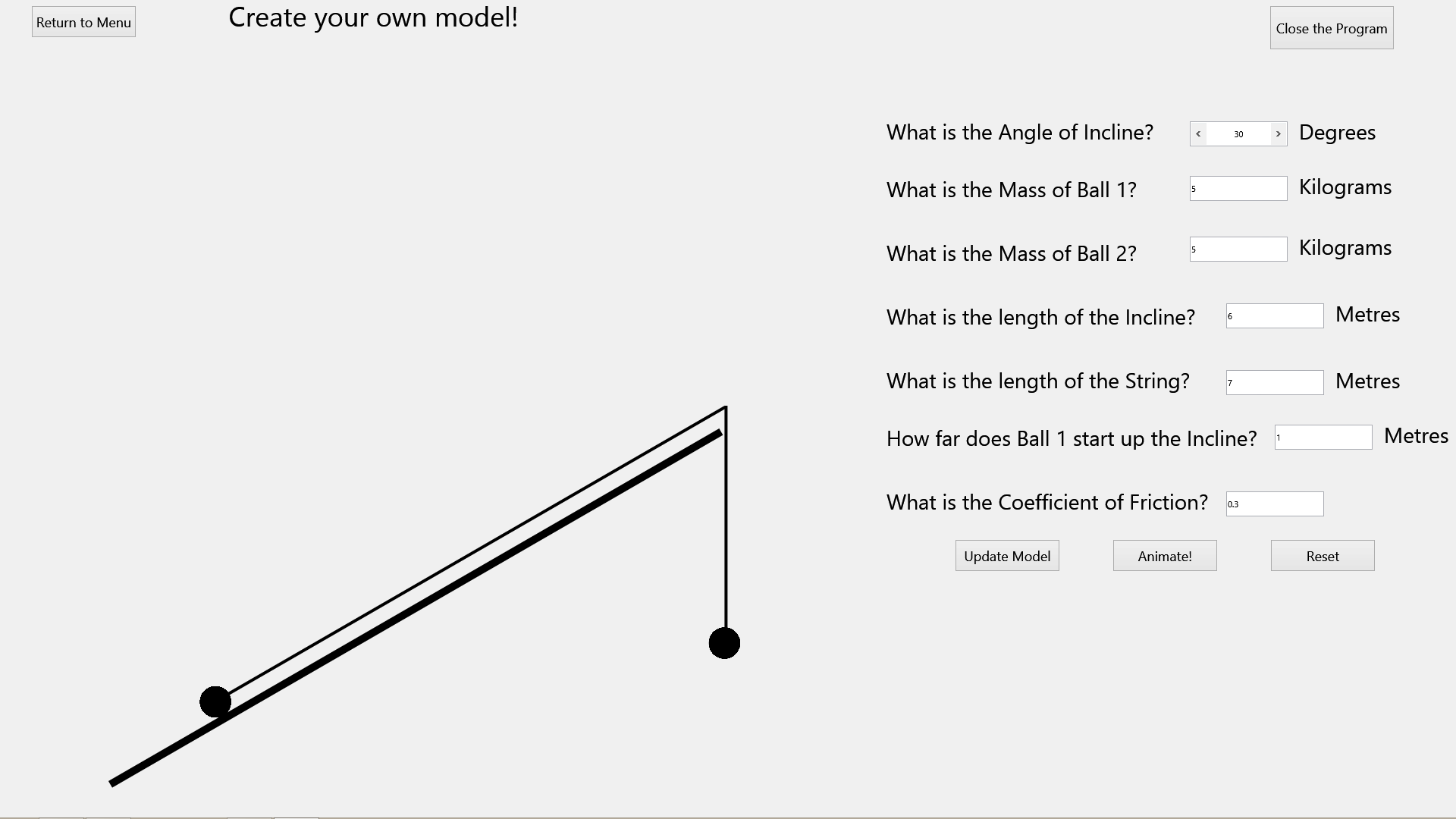
#### Create your own Model Clean



#### Create your own Model Annotated

CloseProgramButton

OwnModelFormWelcomeMessage



RequestForCoefficientOfFriction

RequestDistanceBall1StartsUpPlank

RequestForLengthOfString

RequestForLengthOfPlank

RequestMassOfBall2

RequestMassOfBall1

RequestForPlankAngle

AngleOfPlankSelector

MassOfBall2Selector

MassOfBall1Selector

LengthOfPlankSelector

Selector

Selector

LengthOfStringSelector

Selector

HowFarBall1UpPlankSelector

CoefficientOfFrictionSelector

HowFarBall1UpPlankMetres

StringLengthMetres

PlankLengthMetres

Ball2Kilograms

Ball1Kilograms

AngleDegrees

VerticalString

PlankString

Plank

Ball2

Ball1

ResetButton

AnimateModelButton

UpdateModelButton

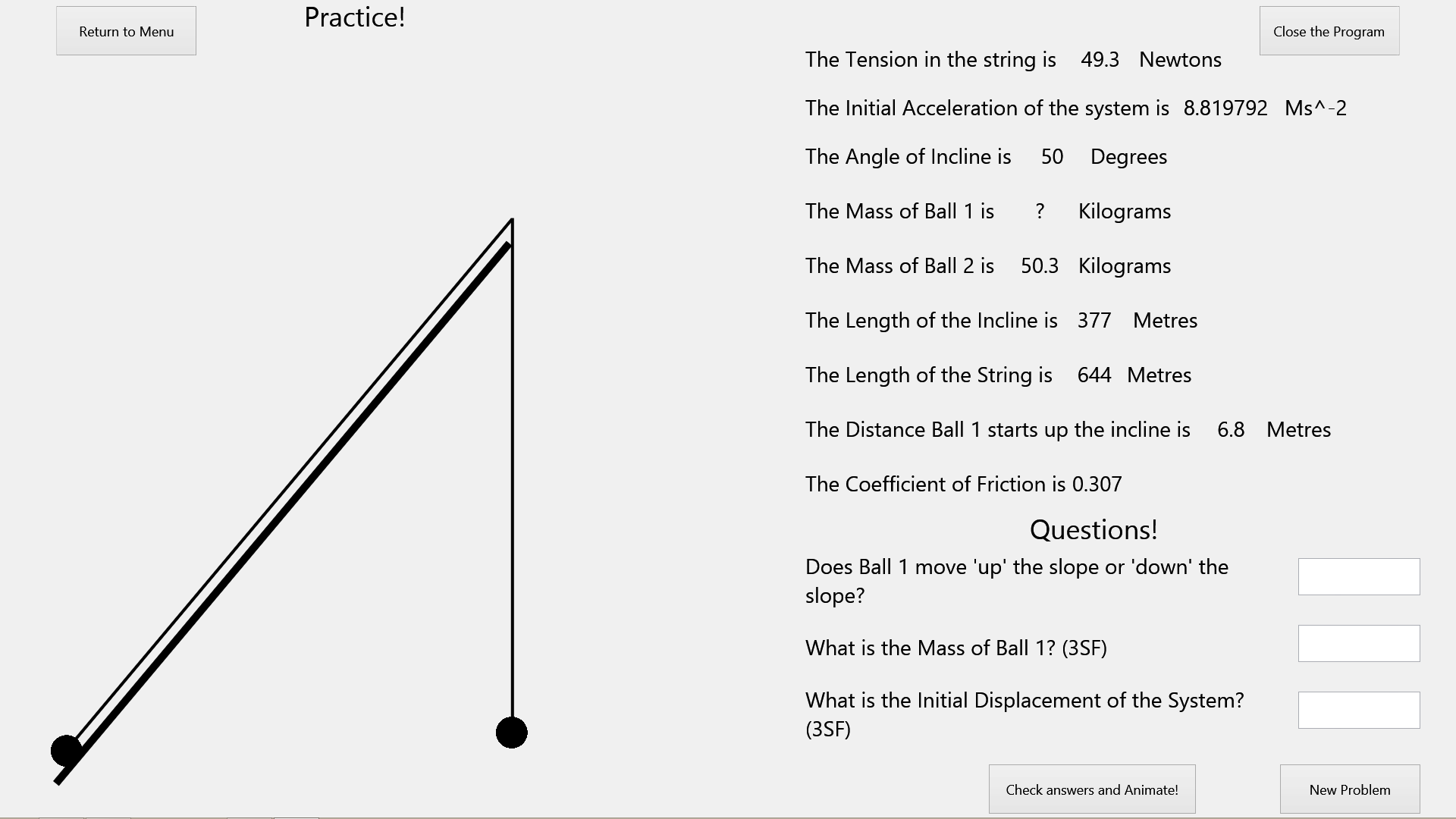
ReturnToMenuButton

#### Create your own Model Table

|  |  |
| --- | --- |
| **Element** | **Type** |
| OwnModelFormWelcomeMessage | TLabel |
| ReturnToMenuButton | TButton |
| Ball1 | TBall (Programmer Defined) |
| Ball2 | TBall (Programmer Defined) |
| Plank | TLine |
| PlankString | TLine |
| VerticalString | TLine |
| RequestForPlankAngle | TLabel |
| RequestMassOfBall1 | TLabel |
| RequestMassOfBall2 | TLabel |
| RequestForLengthOfPlank | TLabel |
| RequestForLengthOfString | TLabel |
| RequestDistanceBall1StartsUpPlank | TLabel |
| RequestForCoefficientOfFriction | TLabel |
| CloseProgramButton | TButton |
| AngleOfPlankSelector | TSpinBox |
| MassOfBall1Selector | TEdit |
| MassOfBall2Selector | TEdit |
| LengthOfPlankSelector | TEdit |
| LengthOfStringSelector | TEdit |
| HowFarBall1UpPlankSelector | TEdit |
| CoefficientOfFrictionSelector | TEdit |
| UpdateModelButton | TButton |
| AnimateModelButton | TButton |
| ResetButton | TButton |
| AngleDegrees | TLabel |
| Ball1Kilograms | TLabel |
| Ball2Kilograms | TLabel |
| PlankLengthMetres | TLabel |
| StringLengthMetres | TLabel |
| HowFarBall1UpPlankMetres | TLabel |

### Practice Form

#### Practice Clean



CloseProgramButton

PracticeFormWelcomeMessage

#### Practice Annotated

Ball2MassKilograms

Ball1MassKilograms

HowFarBall1StartsUpPlankMetres

LengthOfStringMetres

LengthOfPlankMetres

AngleOfPlankDegrees

TensionNewtons

AccelerationMetresPerSecondSquared

TensionShow

AccelerationShow

AngleOfPlankShow

Ball1MassShow

Ball2MassShow

LengthOfPlankShow

LengthOfStringShow

HowFarBall1StartsUpPlankShow

HowFarBall1StartsUpPlankDisclaimer

CoefficientOfFrictionShow

CoefficientOfFrictionDisclaimer

Question2Answer

Question3Answer

Question1Answer

NewProblemButton

CheckAndAnimateButton

Question2

Question3

Question1

QuestionsDisclaimer

LengthOfStringDisclaimer

LengthOfPlankDisclaimer

Ball1MassDisclaimer

Ball2MassDisclaimer

AngleOfPlankDisclaimer

AccelerationDisclaimer

TensionDisclaimer

VerticalString

Ball1

Plank

Ball2

PlankString

ReturnToMenuButton

|  |  |
| --- | --- |
| **Element** | **Type** |
| ReturnToMenuButton | TButton |
| PlankString | TLine |
| VerticalString | TLine |
| Plank | TLine |
| Ball1 | TBall (Programmer Defined) |
| Ball2 | TBall (Programmer Defined) |
| PracticeFormWelcomeMessage | TLabel |
| TensionDisclaimer | TLabel |
| AccelerationDisclaimer | TLabel |
| AngleOfPlankDisclaimer | TLabel |
| Ball1MassDisclaimer | TLabel |
| Ball2MassDisclaimer | TLabel |
| LengthOfPlankDisclaimer | TLabel |
| LengthOfStringDisclaimer | TLabel |
| HowFarBall1StartsUpPlankDisclaimer | TLabel |
| CoefficientOfFrictionDisclaimer | TLabel |
| Question1 | TLabel |
| Question2 | TLabel |
| Question3 | TLabel |
| CheckandAnimateButton | TButton |
| TensionShow | TLabel |
| AccelerationShow | TLabel |
| AngleOfPlankShow | TLabel |
| Ball1MassShow | TLabel |
| Ball2MassShow | TLabel |
| LengthOfPlankShow | TLabel |
| LengthOfStringShow | TLabel |
| HowFarBall1StartsUpPlankShow | TLabel |
| CoefficientOfFrictionShow | TLabel |
| CloseProgramButton | TButton |
| TensionNewtons | TLabel |
| AccelerationMetresPerSecondSquared | TLabel |
| AngleOfPlankDegrees | TLabel |
| Ball1MassKilograms | TLabel |
| Ball2MassKilograms | TLabel |
| LengthOfPlankMetres | TLabel |
| LengthOfStringMetres | TLabel |
| HowFarBall1StartsUpPlankMetres | TLabel |
| QuestionsDisclaimer | TLabel |
| Question1Answer | TLabel |
| Question2Answer | TLabel |
| Question3Answer | TLabel |
| NewProblemButton | TButton |

#### Practice Table

## Description of System Settings

Due to the nature of the program, some ‘system settings’ are constants that I define in each Unit. Here is a table explaining these settings, and what changing them would do:

|  |  |  |  |
| --- | --- | --- | --- |
| **Setting Name** | **Default Value** | **Units it’s found in** | **What it does and what changing it would effect** |
| g | 9.8 | OwnModelUnit, PracticeUnit | This is the stored approximation of acceleration due to gravity on earth. 9.8 is a fair approximation but if the maintainer wants to make the modelling even more accurate this value can be changed to be more accurate. |
| Smoothness Constant | 100 | OwnModelUnit, PracticeUnit | This is a constant that defines the number of small delta distances of animation will occur to give the perceived effect of acceleration. In essence, the higher this number, the smoother the animations will be. However, too high a number may result in lag due to the number of calculations that have to be executed. In my opinion, 100 is a good balance. However, a maintainer can change this value if they wish. |

The other system setting that could be potentially changed is the fact that the program is by default run in fullscreen. This was done by request of the main user, but can be changed if necessary. However, it has to be changed within each Unit that displays a form. It is normally listed as a setting of the form that can easily be changed.

## Functionality

### Introduction to Functionality

This section will explain how all the units and their subroutines interface with each other during program execution. To access the source code for the program you have to open the project file using a Delphi-Firemonkey compatibly compiler. It is important to note that the nature of Delphi-Firemonkey means that different parts of the program (Forms/Units) have their own sections of code and are modules themselves. To this end, there are no real ‘global variables’ within the program. However, each unit has their own variables which are used throughout the unit. These are much like global variables, except that they are only used by that unit. Therefore, when going through each unit I will refer to these as ‘unit global variables’.

However, there is some code used globally throughout the program. This is the code that defines the programmer-defined TBall class/object and is found in the BallUnit.

Starting with the BallUnit, this guide will now step through each unit explaining the functionality. All program code is in italics.

### BallUnit/TBall Object Overview

This Unit is only used to define the TBall object. This object is used by other units to display a Ball on screen.

#### Uses

The Program code below shows the System/FMX resources that the Unit uses:

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,FMX.Objects;*

#### Class Declarations

The only class declaration made is to declare and define the TBall object:

*type*

*TBall = Class(TImage)*

*Private*

*Public*

*Constructor Create(Stream:TStream; var AOwner);*

*Destructor Destroy;*

*End;*

#### Unit Global Variables

There are no Unit Global Variables for this unit.

#### Constants

There are no constants for this unit.

#### Explanation of each subroutine’s functionality

##### Constructor TBall.Create(Stream:TStream;Var AOwner)

This Constructor is part of the TBall class and is used to create the Ball on screen.

##### Destructor TBall.Destroy

This Destructor is part of the TBall class and is used to destroy the object so that memory is freed up.

#### Explanation of modular structure of this Unit’s code and how the code works

##### Creating a Ball on screen:

To create a ball on screen, the TBall.Create Constructor is called. This subroutines doesn’t call any other subroutines during execution:

*Constructor TBall.Create(Stream: TStream; Var AOwner);*

*begin*

*Inherited Create(TComponent(AOwner));*

*Bitmap.LoadFromStream(Stream);*

*Parent := TFMXObject(AOwner);*

*end;*

##### Freeing a ball from memory:

To free a ball from memory, the TBall.Destroy Destructor is called. This subroutine doesn’t call any other subroutines during execution.

*Destructor TBall.Destroy;*

*begin*

*Inherited Destroy;*

*end;*

### MainMenuUnit

This unit displays a form during execution which allows the user to move between the program’s forms.

#### Uses

The Program code below shows the System/FMX resources/other units that the Unit uses. This unit uses OwnModelUnit and PracticeUnit so that it has access to OwnModelForm and PracticeForm. This allows the MainMenuUnit to show and hide OwnModelForm and PracticeForm:

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls, OwnModelUnit, PracticeUnit;*

#### Class Declarations

The only class declaration made is to declare and define the TMainMenuForm class:

*type*

*TMainMenuForm = class(TForm)*

*GoToOwnModelButton: TButton;*

*GoToPracticeButton: TButton;*

*WelcomeMessage: TLabel;*

*CloseProgramButton: TButton;*

*procedure GoToOwnModelButtonClick(Sender: TObject);*

*procedure GoToPracticeButtonClick(Sender: TObject);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*private*

*public*

*end;*

#### Unit Global Variables

This form has one Unit global variable, the Main Menu Form:

*var*

*MainMenuForm: TMainMenuForm;*

#### Constants

There are no constants for this unit.

#### Explanation of each subroutine’s functionality

##### procedure TMainMenuForm.CloseProgramButtonClick(Sender: TObject);

This procedure closes the program.

##### procedure TMainMenuForm.GoToOwnModelButtonClick(Sender: TObject);

This procedure changes the program so it is now displaying the OwnModelForm.

##### procedure TMainMenuForm.GoToPracticeButtonClick(Sender: TObject);

This procedure changes the program so it is now displaying the PracticeForm.

#### Explanation of modular structure of this Unit’s code and how the code works (With references to Design and Testing)

##### Closing the program

This Functionality was tested in Test 2 of the System Testing and passed.

To do this, the CloseProgramButton is pressed. This event triggers this procedure to be executed, halting the program:

*procedure TMainMenuForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*end;*

##### Moving to the OwnModelForm

This Functionality was tested in Test 3 of the System Testing and passed.

To do this, the GoToOwnModelButton is pressed. This event triggers this procedure to be executed, hiding the MainMenuForm and showing the OwnModelForm.

*procedure TMainMenuForm.GoToOwnModelButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Hide;*

*OwnModelForm.Show;*

*end;*

##### Moving to the PracticeForm

This Functionality was tested in Test 4 of the System Testing and passed.

To do this, the GoToPracticeButton is pressed. This event triggers this procedure to be executed, hiding the MainMenuForm and showing the PracticeForm.

*procedure TMainMenuForm.GoToPracticeButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Hide;*

*PracticeForm.Show;*

*end;*

### OwnModelUnit

This Unit displays a form which allows the user to create their own M1 ‘pulley systems on an incline’ model.

#### Uses

The Program code below shows the System/FMX resources/libraries/other units that the Unit uses. This unit uses the Math library so it can use the built in Sin,Cos and DegtoRad functions. The unit also uses BallUnit so that it has access to the TBall class:

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,*

*FMX.Objects, Math, FMX.Edit,BallUnit;*

The unit also uses the MainMenuUnit at the implementation clause to give the unit access to the MainMenuForm. This allows the unit to show the MainMenuForm:

*implementation*

*uses MainMenuUnit;*

#### Class Declarations

The only class declaration made is to declare and define the TOwnModelForm class:

*type*

*TOwnModelForm = class(TForm)*

*Plank: TLine;*

*OwnModelFormWelcomeMessage: TLabel;*

*RequestForPlankAngle: TLabel;*

*RequestMassofBall1: TLabel;*

*RequestMassofBall2: TLabel;*

*RequestForLengthofPlank: TLabel;*

*RequestforDistanceBall1StartsupPlank: TLabel;*

*RequestforLengthofString: TLabel;*

*RequestforCoefficientOfFriction: TLabel;*

*AngleOfPlankSelector: TSpinBox;*

*AngleDegrees: TLabel;*

*MassOfBall1Selector: TEdit;*

*Ball1Kilograms: TLabel;*

*MassOfBall2Selector: TEdit;*

*LengthOfPlankSelector: TEdit;*

*LengthOfStringSelector: TEdit;*

*HowFarBall1UpPlankSelector: TEdit;*

*CoefficientOfFrictionSelector: TEdit;*

*Ball2Kilograms: TLabel;*

*PlankLengthMetres: TLabel;*

*StringLengthMetres: TLabel;*

*HowFarBall1UpPlankMetres: TLabel;*

*PlankString: TLine;*

*VerticalString: TLine;*

*AnimateModelButton: TButton;*

*UpdateModelButton: TButton;*

*ResetButton: TButton;*

*CloseProgramButton: TButton;*

*ReturnToMenuButton: TButton;*

*procedure AngleOfPlankSelectorChange(Sender: TObject);*

*procedure MassOfBall1SelectorChange(Sender: TObject);*

*procedure MassOfBall2SelectorChange(Sender: TObject);*

*procedure LengthOfPlankSelectorChange(Sender: TObject);*

*procedure LengthOfStringSelectorChange(Sender: TObject);*

*procedure HowFarBall1UpPlankSelectorChange(Sender: TObject);*

*procedure CoefficientOfFrictionSelectorChange(Sender: TObject);*

*procedure AnimatePlankAngle;*

*procedure AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*procedure AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX : real);*

*procedure AnimateBallsToCorrectPositions;*

*Procedure AnimateStringToCorrectHeight;*

*procedure UpdateModel;*

*procedure AnimateModelButtonClick(Sender: TObject);*

*procedure FormCreate(Sender: TObject);*

*procedure ResetToDefaultValues;*

*procedure UpdateModelButtonClick(Sender: TObject);*

*procedure CheckChosenValues;*

*procedure ResetButtonClick(Sender: TObject);*

*procedure FormKeyUp(Sender: TObject; var Key: Word; var KeyChar: Char;*

*Shift: TShiftState);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*procedure ReturnToMenuButtonClick(Sender: TObject);*

*procedure FormShow(Sender: TObject);*

*private*

*public*

*end;*

#### Unit Global Variables

This unit uses one global variable for the OwnModelForm, and two for the two Balls to be displayed on screen (members of the TBall class from the BallUnit). Moreover, there are global variables for many mathematical quantities used throughout the unit. Their use is obvious from their identifiers:

*var*

*OwnModelForm: TOwnModelForm;*

*PlankAngleDegrees, PlankAngleRadians, Ball1Mass, Ball2Mass, PlankLength, StringLength, DistanceBall1StartsUpPlank, CoefficientOfFriction, DefaultPlankAngleDegrees, DefaultBall1Mass, DefaultBall2Mass, DefaultPlankLength, DefaultStringLength, DefaultDistanceBall1StartsUpPlank, DefaultCoefficientOfFriction,HeightOfPlankPixels : real;*

*Ball1,Ball2: TBall;*

#### Constants

This unit uses two constants. One to represent the value for natural freefall acceleration on earth (g or 9.8) and one which is called ‘SmoothnessConstant’. SmoothnessConstant is a programmer chosen number that effects how smooth the animation occurs. The bigger, the smoother the animation. 100 seems to be sufficient but it can be increased higher for a smooth animation:

*CONST*

*g = 9.8;*

*SmoothnessConstant=100;*

#### Explanation of each subroutine’s functionality

##### function ConvertMetresToPixels(Input : real) : real;

This function takes an input in metres and outputs the number of pixels that length represents on the screen.

##### function CalculateHeightofPlankToUseInPixels : real;

This function works out how many pixels the plank should take up on the screen based on the size of the display.

##### function CalculateSignificantFigures(Input:String) : integer;

This function takes a real number represented as a string and outputs the number of significant figures it has.

##### function RoundInputasString(Input:String;EndLength:Integer) : string;

This function takes a real number with no decimal place represented as a string and rounds it to a chosen number(EndLength) of significant figures.

##### function MakeXDigits(Input:real;X:Integer) : real;

This function takes a real number and outputs that number to a chosen number(X) of digits in length.

##### function CalculateIfValidXDigitPositiveReal(Input:String; X:integer) : boolean;

This function checks if a real number represented as a string is positive and less than or equal to X digits in length.

##### function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;

This function checks calculates the Y coordinate of the Top of the Plank.

##### function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;

This function checks calculates the X coordinate of the Top of the Plank.

##### procedure TOwnModelForm.AnimateBallsToCorrectPositions;

This procedure animates the balls on screen to the correct position based on user input.

##### procedure TOwnModelForm.AnimateModelButtonClick(Sender: TObject);

This procedure animates the model based on its configuration.

##### procedure TOwnModelForm.AnimatePlankAngle;

This procedure animates the plank angle correctly based on user input.

##### procedure TOwnModelForm.AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);

This procedure animates the PlankString element to the correct angle and position based on user input.

##### procedure TOwnModelForm.AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);

This procedure animates the VerticalString to the correct position based on user input.

##### procedure TOwnModelForm.AnimateStringToCorrectHeight;

This procedure animates the height of VerticalString and PlankString such that the total length of the string matches user input.

##### procedure TOwnModelForm.UpdateModel;

This procedure updates the model based on user input.

##### procedure TOwnModelForm.UpdateModelButtonClick(Sender: TObject);

This procedure checks that user input is valid before updating the model based on the inputs.

##### procedure TOwnModelForm.AngleOfPlankSelectorChange(Sender: TObject);

This procedure checks that the plank angle chosen by the user is valid before updating the relevant variables.

##### procedure TOwnModelForm.CoefficientOfFrictionSelectorChange(Sender: TObject);

This procedure checks that the Coefficient of Friction chosen by the user is valid before updating the relevant variables.

##### procedure TOwnModelForm.ResetButtonClick(Sender: TObject);

This procedure resets all variable values to defaults before updating the model accordingly.

##### procedure TOwnModelForm.ResetToDefaultValues;

This procedure resets all variable values to defaults.

##### procedure TOwnModelForm.ReturnToMenuButtonClick(Sender: TObject);

This procedure moves the program to the MainMenuForm.

##### procedure TOwnModelForm.CheckChosenValues;

This procedure validates user input based on mathematical rules that must be obeyed for a model to be of M1 standard.

##### procedure TOwnModelForm.CloseProgramButtonClick(Sender: TObject);

This procedure closes the program.

##### procedure TOwnModelForm.FormCreate(Sender: TObject);

This procedure initializes all necessary variables and objects.

##### procedure TOwnModelForm.FormKeyUp(Sender: TObject; var Key: Word; var KeyChar: Char; Shift: TShiftState);

This procedure checks that user input is valid before updating the model based on the inputs.

##### procedure TOwnModelForm.FormShow(Sender: TObject);

This procedure sets the displayed model to its default configuration.

##### procedure TOwnModelForm.HowFarBall1UpPlankSelectorChange(Sender: TObject);

This procedure checks that the distance ball 1 starts up the plank chosen by the user is valid before updating the relevant variables.

##### procedure TOwnModelForm.LengthOfPlankSelectorChange(Sender: TObject);

This procedure checks that the Plank Length chosen by the user is valid before updating relevant variables.

##### procedure TOwnModelForm.LengthOfStringSelectorChange(Sender: TObject);

This procedure checks that the String Length chosen by the user is valid before updating relevant variables.

##### procedure TOwnModelForm.MassOfBall1SelectorChange(Sender: TObject);

This procedure checks that the Mass of Ball 1 chosen by the user is valid before updating relevant variables.

##### procedure TOwnModelForm.MassOfBall2SelectorChange(Sender: TObject);

This procedure checks that the Mass of Ball 2 chosen by the user is valid before updating relevant variables.

#### Explanation of modular structure of this Unit’s code and how the code works (With references to Design and Testing)

Note that any highlighted piece of code is a subroutine found in the next section, ‘referenced subroutines’. These are included there as they referenced many times in the main modules found below. Also note any complicated code is annotated and explained.

##### Initializing variables and objects when the form is created (at program start up)

*procedure TOwnModelForm.FormCreate(Sender: TObject);*

*var*

*Ballstream : TResourceStream;*

*begin*

*BallStream := TResourceStream.Create(MainInstance, 'BallImage', RT\_RCDATA);*

*Ball1:=TBall.Create(BallStream,Self);*

*Ball2:=TBall.Create(BallStream,Self);*

*DefaultPlankLength:=6;*

*DefaultPlankAngleDegrees:=30;*

*DefaultBall1Mass:=5;*

*DefaultBall2Mass:=5;*

*DefaultStringLength:=7;*

*DefaultDistanceBall1StartsUpPlank:=1;*

*DefaultCoefficientOfFriction:=0.3;*

*ResetToDefaultValues;*

*end;*

##### Initializing model to default arrangement when the form is shown

This Functionality was tested in Test 3 of the System Testing and passed.

*procedure TOwnModelForm.FormShow(Sender: TObject);*

*var TopOfSlopeY,TopOfSlopeX,BottomOfPlankY,BottomOfPlankX,Ball1X,Ball1Y,Ball2X,Ball2Y : real;*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*Plank.SetBounds(70,OwnModelForm.Height-50-HeightOfPlankPixels,50,HeightofPlankPixels);*

*ResetToDefaultValues;*

*Plank.RotationAngle:=90-PlankAngleDegrees;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*PlankString.rotationangle:=90-PlankAngleDegrees;*

*PlankString.Position.X:=TopOfSlopeX+80;*

*PlankString.Position.Y:=TopOfSlopeY-30;*

*VerticalString.Position.X:=TopOfSlopeX+80;*

*VerticalString.Position.Y:=TopOfSlopeY-30;*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*Ball1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*Ball1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*Ball2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*Ball2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.Position.X:=Ball1X;*

*Ball1.Position.Y:=Ball1Y;*

*Ball2.Position.X:=Ball2X;*

*Ball2.Position.Y:=Ball2Y;*

*PlankString.Height:=ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank);*

*VerticalString.Height:=ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank);*

*end;*

##### Changing the Plank Angle

This Functionality was tested in Tests 9-11 of the System Testing and they all passed.

It is done by using the AngleOfPlankSelector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.AngleOfPlankSelectorChange(Sender: TObject);*

*begin*

*if (StrtoInt(AngleofPlankSelector.Text)>89) or (StrtoInt(AngleofPlankSelector.Text)<1)then*

*begin*

*AngleOfPlankSelector.Text:=FloattoStr(DefaultPlankAngleDegrees);*

*ShowMessage('Not a valid angle of incline, please select an integer between 0 and 90');*

*end;*

*PlankAngleDegrees:=StrToInt(AngleofPlankSelector.Text);*

*PlankAngleRadians:=DegtoRad(PlankAngleDegrees);*

*end;*

##### Changing the Mass of Ball 1

This Functionality was tested in Tests 12-14 of the System Testing and they all passed.

It is done by using the MassOfBall1Selector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.MassOfBall1SelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(MassOfBall1Selector.Text,3) = false then*

*begin*

*MassOfBall1Selector.Text:=FloattoStr(DefaultBall1Mass);*

*ShowMessage('Not a valid mass of ball 1, please select a positive 3 digit or less real number');*

*end;*

*Ball1Mass:=StrToFloat(MassOfBall1Selector.Text);*

*end;*

##### Changing the Mass of Ball 2

This Functionality was tested in Tests 15-17 of the System Testing and they all passed.

It is done by using the MassOfBall2Selector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.MassOfBall2SelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(MassOfBall2Selector.Text,3) = false then*

*begin*

*MassOfBall2Selector.Text:=FloattoStr(DefaultBall2Mass);*

*ShowMessage('Not a valid mass of ball 2, please select a positive 3 digit or less real number');*

*end;*

*Ball2Mass:=StrToFloat(MassOfBall2Selector.Text);*

*end;*

##### Changing the Plank Length

This Functionality was tested in Tests 18-20 of the System Testing and they all passed.

It is done by using the LengthOfPlankSelector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.LengthOfPlankSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(LengthOfPlankSelector.Text,3) = false then*

*begin*

*LengthOfPlankSelector.Text:=FloattoStr(DefaultPlankLength);*

*ShowMessage('Not a valid plank length, please select a positive 3 digit or less real number');*

*end;*

*PlankLength:=StrToFloat(LengthOfPlankSelector.Text);*

*end;*

##### Changing the String Length

This Functionality was tested in Tests 21-23 of the System Testing and they all passed.

It is done by using the LengthOfStringSelector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.LengthOfStringSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(LengthOfStringSelector.Text,3) = false then*

*begin*

*LengthOfStringSelector.Text:=FloattoStr(DefaultStringLength);*

*ShowMessage('Not a valid string length, please select a positive 3 digit or less real number');*

*end;*

*StringLength:=StrToFloat(LengthOfStringSelector.Text);*

*end;*

##### Changing the distance Ball 1 starts up the Plank

This Functionality was tested in Tests 24-26 of the System Testing and they all passed.

It is done by using the HowFarBall1UpPlankSelector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.HowFarBall1UpPlankSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(HowFarBall1UpPlankSelector.Text,3) = false then*

*begin*

*HowFarBall1UpPlankSelector.Text:=FloattoStr(DefaultDistanceBall1StartsUpPlank);*

*ShowMessage('Not a valid distance for ball 1 to start up plank, please select a positive 3 digit or less real number');*

*end;*

*DistanceBall1StartsUpPlank:=StrToFloat(HowFarBall1UpPlankSelector.Text);;*

*end;*

##### Changing the Coefficient Of Friction

This Functionality was tested in Tests 27-29 of the System Testing and they all passed.

It is done by using the CoefficientOfFrictionSelector to select a value. When this value is changed, this subroutine is called, validating the input before changing any relevant variables. If the input is invalid, an error message is displayed and the variables are reset to defaults:

*procedure TOwnModelForm.CoefficientOfFrictionSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(CoefficientOfFrictionSelector.Text,4) = false then*

*begin*

*CoefficientOfFrictionSelector.Text:=FloattoStr(DefaultCoefficientOfFriction);*

*ShowMessage('Not a valid coefficient of friction, please select a positive 4 digit or less real number');*

*end;*

*CoefficientOfFriction:=StrToFloat(CoefficientOfFrictionSelector.Text);*

*end;*

##### Updating the model to a user defined configuration (Note Design for Update Model Algorithm is found in Design document)

This Functionality was tested in Tests 30-38 of the System Testing and they all passed.

This is done by pressing the UpdateModelButton or pressing the Return Key. Each event triggers the relevant procedure below, validating the configuration mathematically before animating the model to that configuration:

*procedure TOwnModelForm.UpdateModelButtonClick(Sender: TObject);*

*begin*

*CheckChosenValues;*

*UpdateModel;*

*end;*

*procedure TOwnModelForm.FormKeyUp(Sender: TObject; var Key: Word; var KeyChar: Char; Shift: TShiftState);*

*begin*

*if Key=VKReturn then*

*begin*

*CheckChosenValues;*

*UpdateModel;*

*end;*

*end;*

##### Animating the model based on its current configuration (Annotated. Note Design for Animation Algorithm is found in Design document)

This Functionality was tested in Tests 39-43 of the System Testing and they all passed.

This is done by pressing the AnimateModelButton. This event triggers this procedure, which is very complicated so I have annotated it for clarification:

*procedure TOwnModelForm.AnimateModelButtonClick(Sender: TObject);*

*var UpSlope : string;*

*Counter : integer;*

*NewXBall1,NewYBall1,NewXBall2,NewYBall2,TimeForThisMovement,PartEndVelocity,Ball2YLocation,Ball1XLocation,Ball1YLocation,EachPartOfInitialMovementLength, InitialMovementLength,ExtraDisplacementLength,EachPartOfExtraDisplacementLength,Acceleration,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround,CurrentInitialSpeed,WaitingTime,LowestRatioForMovement,HighestRatioForMovement:real;*

**Using Force analysis to determine which direction Ball 1 will move. As there are 3 options, a string UpSlope is used to store the result.**

**As Animations are handled within Delphi-Firemonkey after all have been calculated, this variable is used to keep track of the time a given animation must wait before executing. would all animate at the same time after all calculations are done.**

*begin*

*CurrentInitialSpeed:=0;*

*WaitingTime:=0;*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope'*

*else UpSlope:= 'Stationery';*

*if UpSlope = 'UpSlope' then*

**If Ball 1 would move up the slope, only particular Maths applies, so its case must be considered individually.**

*begin*

*Acceleration:=((-Ball1Mass\*g\*((Sin(PlankAngleRadians))+(CoefficientofFriction\*(Cos(PlankAngleRadians)))))+(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

**Using more Force analysis as well as constant acceleration formulae, the distance the Balls will originally move and the extra displacement Ball 1 will undertake after Ball 2 hits the floor is calculated. These distances are then divided by SmoothnessConstant to give small delta distances.**

*Ball2YLocation:=Ball2.Position.Y;*

**The initial locations of the balls are loaded into variables.**

**Validation check to make sure remains an M1 problem**

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 1 would fly off the top of the incline')*

*else begin*

**Time for small movement calculated**

**This loop animates the balls a delta initial movement length SmoothnessConstant many times, so the final length the balls have moved is the initial movement length.**

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

**Balls animated to new position**

**New Ball coordinates calculated after delta distance movement.**

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

**Both parts of String animated to new length**

*end;*

**Relevant variables updated for next loop.**

*Ball1XLocation:=NewXBall1;*

**Location of Balls updated after Balls moved initial movement length**

*Ball1YLocation:=NewYBall1;*

*for counter := 1 to (SmoothnessConstant) do*

**This loop animates ball 1 a delta extra displacement length SmoothnessConstant many times up the slope, so the final length the ball has moved is the extra displacement length.**

**Time for small movement calculated**

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

**New Ball 1 coordinates calculated after delta distance movement.**

*NewXBall1:=Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

**Ball 1 animated to new position**

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

**Plank String animated to new length**

**Relevant variables updated for next loop.**

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

**Location of Ball 1 updated after moved extra displacement length up the slope. CurrentInitialSpeed is currently not 0 due to ignoring air resistance but should be under our model, so it is assigned to 0.**

*CurrentInitialSpeed:=0;*

**This loop animates ball 1 a delta extra displacement length SmoothnessConstant many times down the slope, so the final length the ball has moved is the extra displacement length.**

*for counter := 1 to (SmoothnessConstant) do*

**Time for small movement calculated**

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

**New Ball 1 coordinates calculated after delta distance movement.**

*NewXBall1:=Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:=Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

**Ball 1 animated to new position**

**Plank String animated to new length**

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength-(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

**Relevant variables updated for next loop.**

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end*

**If Ball 1 would move down the slope, only particular Maths applies, so its case must be considered individually.**

*else if UpSlope = 'DownSlope' then begin*

*Acceleration:=((Ball1Mass\*g\*((Sin(PlankAngleRadians))-(CoefficientofFriction\*(Cos(PlankAngleRadians)))))-(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*EachPartOfInitialMovementLength: =InitialMovementLength/SmoothnessConstant;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

**Using more Force analysis as well as constant acceleration formulae, the distance the Balls will originally move and the extra displacement Ball 2 will undertake after Ball 1 hits the floor is calculated. These distances are then divided by SmoothnessConstant to give small delta distances.**

*Ball2YLocation:=Ball2.Position.Y;*

**Validation check to make sure remains an M1 problem**

**The initial locations of the balls are loaded into variables.**

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if (PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength\*Sin(PlankAngleRadians)*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 2 would move higher than the top of the incline')*

*else begin*

**This loop animates the balls a delta initial movement length SmoothnessConstant many times, so the final length the balls have moved is the initial movement length.**

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

**Time for small movement calculated**

**New Ball coordinates calculated after delta distance movement.**

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

**Balls animated to new position**

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

**Both parts of String animated to new length**

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

**Relevant variables updated for next loop.**

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

**Y Coordinate of Ball 2 updated after Balls moved initial movement length**

**This loop animates ball 2 a delta extra displacement length SmoothnessConstant many times upwards, so the final length the ball has moved is the extra displacement length.**

**Time for small movement calculated**

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

**New Ball 2 Y coordinate calculated after delta distance movement.**

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

**Ball 2 animated to new position**

**Vertical String animated to new length**

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

**Relevant variables updated for next loop.**

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

**Y Coordinate of Ball 2 updated after moved extra displacement length upwards. CurrentInitialSpeed is currently not 0 due to ignoring air resistance but should be under our model, so it is assigned to 0.**

*CurrentInitialSpeed:=0;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant) do*

**This loop animates ball 2 a delta extra displacement length SmoothnessConstant many times downwards, so the final length the ball has moved is the extra displacement length.**

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

**Time for small movement calculated**

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

**Ball 2 animated to new position**

**New Ball 2 Y coordinate calculated after delta distance movement.**

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength- PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-ExtraDisplacementLength+(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

**Vertical String animated to new length**

*end;*

**Relevant variables updated for next loop.**

*end;*

*end*

**If Ball 1 wouldn’t move, the scenario must be treated independently.**

*else if UpSlope = 'Stationery' then*

**The Ratio’s of Mass of Ball 2 to Mass of Ball 1 that would cause movement are calculated.**

*begin*

*LowestRatioForMovement:= (CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians);*

*HighestRatioForMovement:=(-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians);*

**Case treated independently because a ratio smaller than 0 is impossible**

*if HighestRatioForMovement<0*

**Explanation to user of why no movement**

*then Showmessage('The Ball would not move! For this angle and coefficient of friction, the system will only move where the Mass of Ball 2 divided by the Mass of Ball 1 is greater than ' + FloattoStr(MakeXDigits(LowestRatioForMovement,4)))*

*else Showmessage('The Ball would not move! For this angle and coefficient of friction, the system will only move where the Mass of Ball 2 divided by the Mass of Ball 1 is greater than ' + FloattoStr(MakeXDigits(LowestRatioForMovement,4)) + ' or smaller than ' + FloattoStr(MakeXDigits(HighestRatioForMovement,4)));*

*end;*

**Explanation to user of why no movement if HighestRatioForMovement>0**

*end;*

##### Resetting the Model

This Functionality was tested in Test 44 of the System Testing and passed.

This is done by pressing the ResetButton. This even triggers this procedure, resetting the relevant variables to default values before updating the model:

*procedure TOwnModelForm.ResetButtonClick(Sender: TObject);*

*begin*

*ResetToDefaultValues;*

*UpdateModel;*

*end;*

##### Closing the Program

This Functionality was tested in Test 6 of the System Testing and passed.

This is done by pressing the CloseProgramButton. This event triggers this procedure, halting the program and freeing the ball objects from memory:

*procedure TOwnModelForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*Ball1.Destroy;*

*Ball2.Destroy;*

*end;*

##### Returning to the Main Menu

This Functionality was tested in Test 5 of the System Testing and passed.

This is done by pressing the ReturnToMenuButton. This event triggers this procedure, hiding the OwnModelForm and showing the MainMenuForm.

*procedure TOwnModelForm.ReturnToMenuButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Show;*

*OwnModelForm.Hide;*

*end;*

#### Referenced subroutines

Any highlighted code references another subroutine found in this section.

##### ConvertMetresToPixels

*function ConvertMetresToPixels(Input : real) : real;*

*begin*

*result:=Input\*(HeightOfPlankPixels/PlankLength);*

*end;*

##### CalculateHeightofPlankToUseInPixels

*function CalculateHeightofPlankToUseInPixels : real;*

*begin*

*if OwnModelForm.Height < OwnModelForm.Width then result:= OwnModelForm.Height-150*

*else result:=OwnModelForm.Width-150;*

*end;*

##### CalculateSignificantFigures

*function CalculateSignificantFigures(Input:String) : integer;*

*var Stepper : integer;*

*begin*

*Result:=Length(Input);*

*for Stepper := 1 to Length(Input) do*

*begin*

*if Input[Stepper] = '.' then Result:=Length(Input) - 1*

*end;*

*end;*

##### RoundInputasString

*function RoundInputasString(Input:String;EndLength:Integer) : string;*

*var Stepper : integer;*

*Finished : boolean;*

*begin*

*if ord(Input[Endlength+1])>52*

*then begin*

*if ord(Input[Endlength])<57*

*then Input[EndLength]:=Char((Ord(Input[Endlength]))+1)*

*else begin*

*Input[Endlength]:='0';*

*Stepper:=EndLength-1;*

*while not Finished do*

*begin*

*if ord(input[Stepper])<57*

*then begin*

*Input[Stepper]:=Char((Ord(Input[Stepper]))+1);*

*Finished:=True;*

*end*

*else begin*

*Input[Stepper]:='0';*

*Stepper:=Stepper-1;*

*end;*

*end;*

*end;*

*end;*

*result:=Input;*

*end;*

##### MakeXDigits

*function MakeXDigits(Input:real;X:Integer) : real;*

*Var InputasString,InputasStringNoDecimal : String;*

*DecimalPlace : boolean;*

*Stepper,Counter,i,DecimalPosition : integer;*

*begin*

*InputasString:=FloattoStr(Input);*

*for Stepper := 1 to Length(InputAsString) do*

*begin*

*if InputasString[Stepper]='.'*

*then begin*

*DecimalPosition:=Stepper;*

*DecimalPlace:=true;*

*end*

*else begin*

*SetLength(InputAsStringNoDecimal,Length(InputAsStringNoDecimal)+1);*

*InputasStringNoDecimal[Length(InputAsStringNoDecimal)]:=InputasString[Stepper]*

*end;*

*end;*

*InputasStringNoDecimal:=RoundInputasString(InputasStringNoDecimal,X);*

*SetLength(InputasStringNoDecimal,X);*

*InputasString:=InputasStringNoDecimal;*

*if DecimalPlace=true*

*then begin*

*SetLength(InputasString,X+1);*

*for I := Length(InputasString) downto DecimalPosition+1 do*

*begin*

*InputasString[i]:=InputasString[i-1]*

*end;*

*InputasString[DecimalPosition]:='.';*

*end;*

*Result:=StrtoFloat(InputasString);*

*end;*

##### CalculateIfValidXDigitPositiveReal

*function CalculateIfValidXDigitPositiveReal(Input:String; X:integer) : boolean;*

*begin*

*if (StrtoFloat(input)<0) or (CalculateSignificantFigures(Input)>X) then result:=false*

*else result := true;*

*end;*

##### CalculateTopOfSlopeY

*function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=OwnModelForm.Height-50-(HeightOfPlankPixels\*Cos((pi/2)-PlankAngleRadians));*

*end;*

##### CalculateTopOfSlopeX

*function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=70+(HeightOfPlankPixels\*Sin((pi/2)-PlankAngleRadians));*

*end;*

##### AnimateBallsToCorrectPositions

*procedure TOwnModelForm.AnimateBallsToCorrectPositions;*

*var NewBall1X,NewBall1Y,NewBall2X,NewBall2Y,BottomOfPlankX,BottomOfPlankY : real;*

*begin*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*NewBall1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*NewBall1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*NewBall2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*NewBall2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.AnimateFloat('position.X',NewBall1X,1);*

*Ball1.AnimateFloat('position.Y',NewBall1Y,1);*

*Ball2.AnimateFloat('position.X',NewBall2X,1);*

*Ball2.AnimateFloat('position.Y',NewBall2Y,1);*

*end;*

##### AnimatePlankAngle

*procedure TOwnModelForm.AnimatePlankAngle;*

*begin*

*Plank.AnimateFloat('rotationangle',90-PlankAngleDegrees,1);*

*end;*

##### AnimatePlankStringAngleAndPosition

*procedure TOwnModelForm.AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*begin*

*PlankString.AnimateFloat('rotationangle',90-PlankAngleDegrees,1);*

*PlankString.AnimateFloat('position.X',TopOfSlopeX+80,1);*

*PlankString.AnimateFloat('position.Y',TopOfSlopeY-30,1);*

*end;*

##### AnimateVerticalStringPosition

*procedure TOwnModelForm.AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);*

*begin*

*VerticalString.AnimateFloat('position.X',TopOfSlopeX+80,1);*

*VerticalString.AnimateFloat('position.Y',TopOfSlopeY-30,1);*

*end;*

##### AnimateStringToCorrectHeight

*procedure TOwnModelForm.AnimateStringToCorrectHeight;*

*begin*

*PlankString.AnimateFloat('Height',ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank),1);*

*VerticalString.AnimateFloat('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank),1);*

*end;*

##### UpdateModel

*procedure TOwnModelForm.UpdateModel;*

*var TopOfSlopeY, TopOfSlopeX : real;*

*begin*

*AnimatePlankAngle;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX);*

*AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX);*

*AnimateBallsToCorrectPositions;*

*AnimateStringToCorrectHeight;*

*end;*

##### ResetToDefaultValues

*procedure TOwnModelForm.ResetToDefaultValues;*

*begin*

*PlankLength:=DefaultPlankLength;*

*PlankAngleDegrees:= DefaultPlankAngleDegrees;*

*Ball1Mass:=DefaultBall1Mass;*

*Ball2Mass:=DefaultBall2Mass;*

*StringLength:=DefaultStringLength;*

*DistanceBall1StartsUpPlank:=DefaultDistanceBall1StartsUpPlank;*

*CoefficientOfFriction:=DefaultCoefficientOfFriction;*

*LengthOfPlankSelector.Text:=FloattoStr(DefaultPlankLength);*

*AngleOfPlankSelector.Text:= FloattoStr(DefaultPlankAngleDegrees);*

*MassOfBall1Selector.Text:=FloattoStr(DefaultBall1Mass);*

*MassOfBall2Selector.Text:=FloattoStr(DefaultBall2Mass);*

*LengthOfStringSelector.Text:=FloattoStr(DefaultStringLength);*

*HowFarBall1UpPlankSelector.Text:=FloattoStr(DefaultDistanceBall1StartsUpPlank);*

*CoefficientOfFrictionSelector.Text:=FloattoStr(DefaultCoefficientOfFriction);*

*end;*

##### CheckChosenValues

*procedure TOwnModelForm.CheckChosenValues;*

*begin*

*if DistanceBall1StartsUpPlank+StringLength<PlankLength*

*then begin*

*ShowMessage('Not an accepeted change, this would result in the string not begin long enough to connect two particles in this plane. Resetting to Default Values.');*

*ResettoDefaultValues;*

*end;*

*if DistanceBall1StartsUpPlank>PlankLength*

*then begin*

*ShowMessage('Not an accepted change, the distance Ball 1 starts up the incline cannot be greater than the length of the plank. Resetting to Default Values.');*

*ResetToDefaultValues;*

*end;*

*if StringLength-PlankLength+DistanceBall1StartsUpPlank>PlankLength\*Sin(PlankAngleRadians)*

*then begin*

*ShowMessage('Not an accepted change, this would result in a slack string due to Ball 2 not being able to fall through the floor. Resetting to Default Values');*

*ResetToDefaultValues;*

*end;*

*end;*

### PracticeUnit

This Unit displays a form which questions the user on a random ‘pulley systems on an incline’ M1 model.

#### Uses

The Program code below shows the System/FMX resources/libraries/other units that the Unit uses. This unit uses the Math library so it can use the built in Sin,Cos, DegtoRad and Random functions. The unit also uses BallUnit so that it has access to the TBall class:

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,*

*FMX.Objects,Math,BallUnit, FMX.Edit;*

The unit also uses the MainMenuUnit at the implementation clause to give the unit access to the MainMenuForm. This allows the unit to show the MainMenuForm:

*implementation*

*uses MainMenuUnit;*

#### Class Declarations

The only class declaration made is to declare and define the TPraticeForm class:

*type*

*TPracticeForm = class(TForm)*

*PracticeFormWelcomeMessage: TLabel;*

*Plank: TLine;*

*PlankString: TLine;*

*VerticalString: TLine;*

*TensionDisclaimer: TLabel;*

*AngleOfPlankDisclaimer: TLabel;*

*Ball1MassDisclaimer: TLabel;*

*Ball2MassDisclaimer: TLabel;*

*LengthOfPlankDisclaimer: TLabel;*

*LengthOfStringDisclaimer: TLabel;*

*HowFarBall1StartsUpPlankDisclaimer: TLabel;*

*CoefficientOfFrictionDisclaimer: TLabel;*

*AccelerationDisclaimer: TLabel;*

*TensionNewtons: TLabel;*

*AccelerationMetresPerSecondSquared: TLabel;*

*AngleOfPlankDegrees: TLabel;*

*Ball1MassKilograms: TLabel;*

*Ball2MassKilograms: TLabel;*

*LengthOfPlankMetres: TLabel;*

*LengthOfStringMetres: TLabel;*

*HowFarBall1UpPlankMetres: TLabel;*

*TensionShow: TLabel;*

*AccelerationShow: TLabel;*

*AngleOfPlankShow: TLabel;*

*Ball1MassShow: TLabel;*

*CoefficientOfFrictionShow: TLabel;*

*HowFarBall1UpPlankShow: TLabel;*

*LengthOfStringShow: TLabel;*

*LengthOfPlankShow: TLabel;*

*Ball2MassShow: TLabel;*

*QuestionsDisclaimer: TLabel;*

*Question1: TLabel;*

*Question2: TLabel;*

*Question3: TLabel;*

*CheckAndAnimateButton: TButton;*

*Question1Answer: TEdit;*

*Question2Answer: TEdit;*

*Question3Answer: TEdit;*

*NewProblemButton: TButton;*

*CloseProgramButton: TButton;*

*ReturnToMenuButton: TButton;*

*procedure FormResize(Sender: TObject);*

*procedure UpdateModel;*

*procedure ChangeBallsToCorrectPositions;*

*procedure ChangePlankAngle;*

*procedure ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX:Real);*

*procedure ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX:Real);*

*procedure ChangeStringToCorrectHeight;*

*procedure FormCreate(Sender: TObject);*

*procedure CheckAndAnimateButtonClick(Sender: TObject);*

*procedure NewProblemButtonClick(Sender: TObject);*

*procedure CreateProblem;*

*procedure FormShow(Sender: TObject);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*procedure ReturnToMenuButtonClick(Sender: TObject);*

*private*

*public*

*end;*

#### Unit Global Variables

This unit uses one global variable for the OwnModelForm, and two for the two Balls to be displayed on screen (members of the TBall class from the BallUnit). Moreover, there are global variables for many mathematical quantities used throughout the unit. Their use is obvious from their identifiers. There are also two integers which are used to stored random numbers to aid with generating the random model and a string ‘UpSlope’ which acts as a 3-way Boolean variable to store whether the model remains stationary, moves up the slope or moves down the slope:

*var*

*PracticeForm: TPracticeForm;*

*Acceleration,Tension, PlankAngleDegrees, PlankAngleRadians, Ball1Mass, Ball2Mass, PlankLength, StringLength, DistanceBall1StartsUpPlank, CoefficientOfFriction, HeightOfPlankPixels,InitialMovementLength,ExtraDisplacementLength : real;*

*UpSlope : String;*

*RandomInteger1,RandomInteger2 : integer;*

*Ball1,Ball2: TBall;*

#### Constants

This unit uses two constants. One to represent the value for natural freefall acceleration on earth (g or 9.8) and one which is called ‘SmoothnessConstant’. SmoothnessConstant is a programmer chosen number that effects how smooth the animation occurs. The bigger, the smoother the animation. 100 seems to be sufficient but it can be increased higher for a smooth animation:

*CONST*

*g = 9.8;*

*SmoothnessConstant=100;*

#### Explanation of each subroutine’s functionality

##### function ConvertMetresToPixels(Input : real) : real;

This function takes an input in metres and outputs the number of pixels that length represents on the screen.

##### function CalculateHeightofPlankToUseInPixels : real;

This function works out how many pixels the plank should take up on the screen based on the size of the display.

##### function CalculateSignificantFigures(Input:String) : integer;

This function takes a real number represented as a string and outputs the number of significant figures it has.

##### function Random3DigitReal : real;

This function outputs a random 3 digit real number.

##### function RoundInputasString(Input:String;EndLength:Integer) : string;

This function takes a real number with no decimal place represented as a string and rounds it to a chosen number(EndLength) of significant figures.

##### function MakeXDigits(Input:real;X:Integer) : real;

This function takes a real number and outputs that number to a chosen number(X) of digits in length.

##### function CalculateTension:real;

This function outputs the tension in the string.

##### procedure CreateRandomValues;

This procedure assigns random values to all variables that effect the arrangement of the model.

##### function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;

This function checks calculates the Y coordinate of the Top of the Plank.

##### function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;

This function checks calculates the X coordinate of the Top of the Plank.

##### function SensibleM1Problem : boolean;

This function outputs a Boolean result true if the current model is M1 standard, and false if it isn’t.

##### procedure TPracticeForm.ChangeBallsToCorrectPositions;

This procedure moves the balls to the correct position based on the current values of the relevant variables.

##### procedure TPracticeForm.ChangePlankAngle;

This procedure changes the Plank’s angle to the correct orientation based on the current value of the relevant variable.

##### procedure TPracticeForm.ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);

This procedure changes the angle and position of the PlankString to the correct orientation based on the current values of the relevant variables.

##### procedure TPracticeForm.ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);

This procedure changes the position of the VerticalString to the correct position based on the current values of the relevant variables.

##### procedure TPracticeForm.CheckAndAnimateButtonClick(Sender: TObject);

This procedure checks that the user has answered the questions correctly, informing them if any are wrong. If the answers are correct, the procedure will animate the model based on its current configuration.

##### procedure TPracticeForm.CloseProgramButtonClick(Sender: TObject);

This procedure closes the program.

##### procedure TPracticeForm.ChangeStringToCorrectHeight;

This procedure changes the height of PlankString and VerticalString so the total length of the String matches the relevant variable values.

##### procedure TPracticeForm.UpdateModel;

This procedure updates the configuration of the model based on the relevant variable values.

##### procedure ResetValuestoZero;

This procedure resets all variables that effect the model to 0.

##### procedure TPracticeForm.CreateProblem;

This procedure creates a random model and the questions to be used. It then displays the model.

##### procedure TOwnModelForm.FormCreate(Sender: TObject);

This procedure initializes all necessary variables and objects. The randomize table is also initialized.

##### procedure TOwnModelForm.FormShow(Sender: TObject);

This procedure sets the displayed model to a random configuration when the form is shown.

##### procedure TPracticeForm.NewProblemButtonClick(Sender: TObject);

This procedure creates a random model and the questions to be used. It then displays the model.

##### procedure TPracticeForm.ReturnToMenuButtonClick(Sender: TObject);

This procedure returns the user to the MainMenuForm.

#### Explanation of modular structure of this Unit’s code and how the code works (With references to Design and Testing)

Note that any highlighted piece of code is a subroutine found in the next section, ‘referenced subroutines’. These are included there as they referenced many times in the main modules found below. Also note any complicated code is annotated and explained.

##### Initializing variables, objects and randomize table when the form is created (at program start up)

*procedure TPracticeForm.FormCreate(Sender: TObject);*

*var BallStream:TResourceStream;*

*begin*

*BallStream := TResourceStream.Create(MainInstance, 'BallImage', RT\_RCDATA);*

*Ball1:=TBall.Create(BallStream,Self);*

*Ball2:=TBall.Create(BallStream,Self);*

*Randomize;*

*end;*

##### Initializing model to a random arrangement when the form is shown

This Functionality was tested in Test 4 of the System Testing and passed.

*procedure TPracticeForm.FormShow(Sender: TObject);*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*Plank.SetBounds(70,PracticeForm.Height-50-HeightOfPlankPixels,50,HeightofPlankPixels);*

*CreateProblem;*

*end;*

##### Returning to the Main Menu

This Functionality was tested in Test 7 of the System Testing and passed.

When the ReturnToMenuButton is pressed, the event triggers this procedure. This hides the PracticeForm, and shows the MainMenuForm:

*procedure TPracticeForm.ReturnToMenuButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Show;*

*PracticeForm.Hide;*

*end;*

##### Closing the Program

This Functionality was tested in Test 8 of the System Testing and passed.

This is done by pressing the CloseProgramButton. This event triggers this procedure, halting the program and freeing the ball objects from memory:

*procedure TPracticeForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*Ball1.Destroy;*

*Ball2.Destroy;*

*end;*

##### Checking answers to questions and animating the model

This Functionality was tested in Tests 45-46 of the System Testing and they both passed.

When the CheckAndAnimateButton is pressed, the event triggers this procedure. It first checks that the answers entered in the relevant text boxes are correct. If they are incorrect, the ones which are incorrect are notified to the user. If they are all correct, the model begins to animate. For a better explanation of how the animation functionality works, see the explanation of the relevant code in OwnModelUnit:

*procedure TPracticeForm.CheckAndAnimateButtonClick(Sender: TObject);*

*var Counter : integer;*

*NewXBall1,NewYBall1,NewXBall2,NewYBall2,TimeForThisMovement,PartEndVelocity,Ball2YLocation,Ball1XLocation,Ball1YLocation,EachPartOfInitialMovementLength,EachPartOfExtraDisplacementLength,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround,CurrentInitialSpeed,WaitingTime,LowestRatioForMovement,HighestRatioForMovement:real;*

*Question1Correct,Question2Correct,Question3Correct : boolean;*

*begin*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope';*

*if UpSlope = 'UpSlope' then*

*begin*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*end;*

*if UpSlope = 'DownSlope' then*

*begin*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*end;*

*if (UpSlope='UpSlope') and (Uppercase(Question1Answer.Text)='UP') then Question1Correct:=True;*

*if (UpSlope='DownSlope') and (Uppercase(Question1Answer.Text)='DOWN') then Question1Correct:=True;*

*if (RandomInteger1=1) and (Question2Answer.Text=FloatToStr(Tension)) then Question2Correct:=True;*

*if (RandomInteger1=2) and (Question2Answer.Text=FloatToStr(Acceleration)) then Question2Correct:=True;*

*if (RandomInteger1=3) and (Question2Answer.Text=FloatToStr(PlankAngleDegrees)) then Question2Correct:=True;*

*if (RandomInteger1=4) and (Question2Answer.Text=FloatToStr(Ball1Mass)) then Question2Correct:=True;*

*if (RandomInteger1=5) and (Question2Answer.Text=FloatToStr(Ball2Mass)) then Question2Correct:=True;*

*if (RandomInteger1=6) and (Question2Answer.Text=FloatToStr(CoefficientOfFriction)) then Question2Correct:=True;*

*if (RandomInteger2=1) and (Question3Answer.Text=FloatToStr(Tension)) then Question3Correct:=True;*

*if (RandomInteger2=2) and (Question3Answer.Text=FloatToStr(Acceleration)) then Question3Correct:=True;*

*if (RandomInteger2=3) and (Question3Answer.Text=FloatToStr(PlankAngleDegrees)) then Question3Correct:=True;*

*if (RandomInteger2=4) and (Question3Answer.Text=FloatToStr(Ball1Mass)) then Question3Correct:=True;*

*if (RandomInteger2=5) and (Question3Answer.Text=FloatToStr(Ball2Mass)) then Question3Correct:=True;*

*if (RandomInteger2=6) and (Question3Answer.Text=FloatToStr(CoefficientOfFriction)) then Question3Correct:=True;*

*if (RandomInteger2=7) and (Question3Answer.Text=FloatToStr(InitialMovementLength)) then Question3Correct:=True;*

*if (RandomInteger2=8) and (Question3Answer.Text=FloatToStr(ExtraDisplacementLength)) then Question3Correct:=True;*

*if (Question1Correct=True) and (Question2Correct=True) and (Question3Correct=True)*

*then begin*

*ShowMessage('Answers Correct. Animating!');*

*CurrentInitialSpeed:=0;*

*WaitingTime:=0;*

*if UpSlope = 'UpSlope' then*

*begin*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 1 would fly off the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*CurrentInitialSpeed:=0;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:=Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength-(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end*

*else if UpSlope = 'DownSlope' then begin*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if (PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength\*Sin(PlankAngleRadians)*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 2 would move higher than the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*CurrentInitialSpeed:=0;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-ExtraDisplacementLength+(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end;*

*end*

*else if (Question1Correct=False) and (Question2Correct=True) and (Question3Correct=True) then ShowMessage('Answer to Question 1 is incorrect')*

*else if (Question1Correct=True) and (Question2Correct=False) and (Question3Correct=True) then ShowMessage('Answer to Question 2 is incorrect')*

*else if (Question1Correct=True) and (Question2Correct=True) and (Question3Correct=False) then ShowMessage('Answer to Question 3 is incorrect')*

*else if (Question1Correct=False) and (Question2Correct=False) and (Question3Correct=True) then ShowMessage('Answers to Question 1 and Question 2 are incorrect')*

*else if (Question1Correct=False) and (Question2Correct=True) and (Question3Correct=False) then ShowMessage('Answers to Question 1 and Question 3 are incorrect')*

*else if (Question1Correct=True) and (Question2Correct=False) and (Question3Correct=False) then ShowMessage('Answers to Question 2 and Question 3 are incorrect')*

*else if (Question1Correct=False) and (Question2Correct=False) and (Question3Correct=False) then ShowMessage('Answers to all Questions are incorrect')*

*end;*

##### Generating and displaying a new problem (Note Design for Create Random Problem Algorithm is found in Design document)

This Functionality was tested in Test 47 of the System Testing and passed.

When the NewProblemButton is pressed, the event triggers this procedure. This generates a random M1 ‘Pulley systems on an incline’ problem and displays it. It also generates the questions that will be used and displays them:

*procedure TPracticeForm.NewProblemButtonClick(Sender: TObject);*

*begin*

*CreateProblem;*

*end;*

#### Referenced Subroutines

Any highlighted code references another subroutine found in this section.

##### ConvertMetresToPixels

*function ConvertMetresToPixels(Input : real) : real;*

*begin*

*result:=Input\*(HeightOfPlankPixels/PlankLength);*

*end;*

##### CalculateHeightofPlankToUseInPixels

*function CalculateHeightofPlankToUseInPixels : real;*

*begin*

*if PracticeForm.Height < PracticeForm.Width then result:= PracticeForm.Height-150*

*else result:=PracticeForm.Width-150;*

*end;*

##### CalculateSignificantFigures

*function CalculateSignificantFigures(Input:String) : integer;*

*var Stepper : integer;*

*begin*

*Result:=Length(Input);*

*for Stepper := 1 to Length(Input) do*

*begin*

*if Input[Stepper] = '.' then Result:=Length(Input) - 1*

*end;*

*end;*

##### Random3DigitReal

*function Random3DigitReal : real;*

*var FloatasString : String;*

*Counter,RandomDecisionValue : integer;*

*begin*

*for counter := 1 to 3 do*

*begin*

*FloatasString:=FloatasString+IntToStr(Random(9));*

*end;*

*RandomDecisionValue:=Random(100);*

*if RandomDecisionValue<33 then*

*begin*

*SetLength(FloatAsString,4);*

*FloatAsString[4]:=FloatAsString[3];*

*FloatAsString[3]:=FloatAsString[2];*

*FloatAsString[2]:='.';*

*end;*

*if (RandomDecisionValue>=33) and (RandomDecisionValue<66) then*

*begin*

*SetLength(FloatAsString,4);*

*FloatAsString[4]:=FloatAsString[3];*

*FloatAsString[3]:='.';*

*end;*

*result:=StrtoFloat(FloatasString);*

*end;*

##### RoundInputasString

*function RoundInputasString(Input:String;EndLength:Integer) : string;*

*var Stepper : integer;*

*Finished : boolean;*

*begin*

*if ord(Input[Endlength+1])>52*

*then begin*

*if ord(Input[Endlength])<57*

*then Input[EndLength]:=Char((Ord(Input[Endlength]))+1)*

*else begin*

*Input[Endlength]:='0';*

*Stepper:=EndLength-1;*

*while not Finished do*

*begin*

*if ord(input[Stepper])<57*

*then begin*

*Input[Stepper]:=Char((Ord(Input[Stepper]))+1);*

*Finished:=True;*

*end*

*else begin*

*Input[Stepper]:='0';*

*Stepper:=Stepper-1;*

*end;*

*end;*

*end;*

*end;*

*result:=Input;*

*end;*

##### MakeXDigits

*function MakeXDigits(Input:real;X:Integer) : real;*

*Var InputasString,InputasStringNoDecimal : String;*

*DecimalPlace : boolean;*

*Stepper,Counter,i,DecimalPosition : integer;*

*begin*

*InputasString:=FloattoStr(Input);*

*for Stepper := 1 to Length(InputAsString) do*

*begin*

*if InputasString[Stepper]='.'*

*then begin*

*DecimalPosition:=Stepper;*

*DecimalPlace:=true;*

*end*

*else begin*

*SetLength(InputAsStringNoDecimal,Length(InputAsStringNoDecimal)+1);*

*InputasStringNoDecimal[Length(InputAsStringNoDecimal)]:=InputasString[Stepper]*

*end;*

*end;*

*InputasStringNoDecimal:=RoundInputasString(InputasStringNoDecimal,X);*

*SetLength(InputasStringNoDecimal,X);*

*InputasString:=InputasStringNoDecimal;*

*if DecimalPlace=true*

*then begin*

*SetLength(InputasString,X+1);*

*for I := Length(InputasString) downto DecimalPosition+1 do*

*begin*

*InputasString[i]:=InputasString[i-1]*

*end;*

*InputasString[DecimalPosition]:='.';*

*end;*

*Result:=StrtoFloat(InputasString);*

*end;*

##### CalculateTension

*function CalculateTension:real;*

*var Temp : real;*

*begin*

*if UpSlope='UpSlope' then Temp:=Ball2Mass\*(g-Acceleration);*

*if UpSlope='DownSlope' then Temp:=Ball2Mass\*(g+Acceleration);*

*Result:=MakeXDigits(Temp,4);*

*end;*

##### CreateRandomValues

*procedure CreateRandomValues;*

*begin*

*PlankAngleDegrees:= RandomRange(0,90);*

*PlankAngleRadians:=DegtoRad(PlankAngleDegrees);*

*Ball1Mass:= Random3DigitReal;*

*Ball2Mass:= Random3DigitReal;*

*PlankLength:= Random3DigitReal;*

*DistanceBall1StartsUpPlank:= Random3DigitReal;*

*StringLength:= Random3DigitReal;*

*CoefficientOfFriction:= MakeXDigits(Random,4);*

*end;*

##### CalculateTopOfSlopeY

*function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=PracticeForm.Height-50-(HeightOfPlankPixels\*Cos((pi/2)-PlankAngleRadians));*

*end;*

##### CalculateTopOfSlopeX

*function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=70+(HeightOfPlankPixels\*Sin((pi/2)-PlankAngleRadians));*

*end;*

##### SensibleM1Problem

*function SensibleM1Problem : boolean;*

*var TempAcceleration,InitialMovementLength,ExtraDisplacementLength,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround:real;*

*begin*

*if (DistanceBall1StartsUpPlank+StringLength<PlankLength)*

*or (DistanceBall1StartsUpPlank>PlankLength)*

*or (StringLength-PlankLength+DistanceBall1StartsUpPlank>PlankLength\*Sin(PlankAngleRadians))*

*then result:=false*

*else begin*

*result:=true;*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope'*

*else result:=false;*

*if UpSlope = 'UpSlope' then*

*begin*

*TempAcceleration:=((-Ball1Mass\*g\*((Sin(PlankAngleRadians))+(CoefficientofFriction\*(Cos(PlankAngleRadians)))))+(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*Acceleration:=MakeXDigits(TempAcceleration,7);*

*if Acceleration<0*

*then result:=false*

*else begin*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength then result:=false;*

*end;*

*end*

*else if UpSlope = 'DownSlope' then*

*begin*

*TempAcceleration:=((Ball1Mass\*g\*((Sin(PlankAngleRadians))-(CoefficientofFriction\*(Cos(PlankAngleRadians)))))-(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*Acceleration:=MakeXDigits(TempAcceleration,7);*

*if Acceleration<0*

*then result:=false*

*else begin*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*if PlankLength\*Sin(PlankAngleRadians)-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength > PlankLength\*Sin(PlankAngleRadians) then result:=false;*

*end;*

*end;*

*end;*

*end;*

##### ChangeBallsToCorrectPositions

*procedure TPracticeForm.ChangeBallsToCorrectPositions;*

*var NewBall1X,NewBall1Y,NewBall2X,NewBall2Y,BottomOfPlankX,BottomOfPlankY : real;*

*begin*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*NewBall1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*NewBall1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*NewBall2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*NewBall2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.Position.X:=NewBall1X;*

*Ball1.Position.Y:=NewBall1Y;*

*Ball2.Position.X:=NewBall2X;*

*Ball2.Position.Y:=NewBall2Y;*

*end;*

##### ChangePlankAngle

*procedure TPracticeForm.ChangePlankAngle;*

*begin*

*Plank.RotationAngle:=90-PlankAngleDegrees;*

*end;*

##### ChangePlankStringAngleAndPosition

*procedure TPracticeForm.ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*begin*

*PlankString.RotationAngle:=90-PlankAngleDegrees;*

*PlankString.Position.X:=TopOfSlopeX+6;*

*PlankString.Position.Y:=TopOfSlopeY-30;*

*end;*

##### ChangeVerticalStringPosition

*procedure TPracticeForm.ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);*

*begin*

*VerticalString.Position.X:=TopOfSlopeX+6;*

*VerticalString.Position.Y:=TopOfSlopeY-30;*

*end;*

##### ChangeStringToCorrectHeight

*procedure TPracticeForm.ChangeStringToCorrectHeight;*

*begin*

*PlankString.Height:=ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank);*

*VerticalString.Height:=ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank);*

*end;*

##### UpdateModel

*procedure TPracticeForm.UpdateModel;*

*var TopOfSlopeY,TopOfSlopeX : real;*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*ChangePlankAngle;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX);*

*ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX);*

*ChangeBallsToCorrectPositions;*

*ChangeStringToCorrectHeight;*

*end;*

##### ResetValuestoZero

*procedure ResetValuestoZero;*

*begin*

*RandomInteger1:=0;*

*RandomInteger2:=0;*

*Tension:=0;*

*Acceleration:=0;*

*PlankAngleDegrees:=0;*

*Ball1Mass:=0;*

*Ball2Mass:=0;*

*PlankLength:=0;*

*StringLength:=0;*

*DistanceBall1StartsUpPlank:=0;*

*CoefficientOfFriction:=0;*

*end;*

##### CreateProblem

*procedure TPracticeForm.CreateProblem;*

*Var QuestionsStream : TResourceStream;*

*QuestionsList : TStringList;*

*TensionQuestion,AccelerationQuestion,AngleQuestion,Ball1MassQuestion,Ball2MassQuestion,CoefficientQuestion : boolean;*

*begin*

*Question1Answer.Text:='';*

*Question2Answer.Text:='';*

*Question3Answer.Text:='';*

*ResetValuestoZero;*

*CreateRandomValues;*

*While not SensibleM1Problem = true do CreateRandomValues;*

*Tension:=CalculateTension;*

*QuestionsStream := TResourceStream.Create(MainInstance, 'Questions', RT\_RCDATA);*

*QuestionsList:=TStringList.Create;*

*QuestionsList.LoadFromStream(QuestionsStream);*

*while (RandomInteger1=RandomInteger2)*

*or ((RandomInteger1=6) And (RandomInteger2=4))*

*or ((RandomInteger1=4) And (RandomInteger2=6))*

*or ((RandomInteger1=6) And (RandomInteger2=3))*

*or ((RandomInteger1=3) And (RandomInteger2=6))*

*do begin*

*RandomInteger1:=RandomRange(1,6);*

*RandomInteger2:=RandomRange(1,8);*

*end;*

*case RandomInteger1 of*

*1 : TensionQuestion:=True;*

*2 : AccelerationQuestion:=True;*

*3 : AngleQuestion:=True;*

*4 : Ball1MassQuestion:=True;*

*5 : Ball2MassQuestion:=True;*

*6 : CoefficientQuestion:=True;*

*end;*

*case RandomInteger2 of*

*1 : TensionQuestion:=True;*

*2 : AccelerationQuestion:=True;*

*3 : AngleQuestion:=True;*

*4 : Ball1MassQuestion:=True;*

*5 : Ball2MassQuestion:=True;*

*6 : CoefficientQuestion:=True;*

*end;*

*Question1.Text:=QuestionsList[8];*

*Question2.Text:=QuestionsList[RandomInteger1-1];*

*Question3.Text:=QuestionsList[RandomInteger2-1];*

*if AccelerationQuestion = true then AccelerationShow.Text:='?'*

*else AccelerationShow.Text:=FloattoStr(Acceleration);*

*if TensionQuestion = true then TensionShow.Text:='?'*

*else TensionShow.Text:=FloattoStr(Tension);*

*if AngleQuestion = true then AngleofPlankShow.Text:='?'*

*else AngleOfPlankShow.Text:=FloattoStr(PlankAngleDegrees);*

*if Ball1MassQuestion = true then Ball1MassShow.Text:='?'*

*else Ball1MassShow.Text:=FloattoStr(Ball1Mass);*

*if Ball2MassQuestion = true then Ball2MassShow.Text:='?'*

*else Ball2MassShow.Text:=FloattoStr(Ball2Mass);*

*if CoefficientQuestion = true then CoefficientOfFrictionShow.Text:='?'*

*else CoefficientOfFrictionShow.Text:=FloattoStr(CoefficientOfFriction);*

*LengthOfPlankShow.Text:=FloattoStr(PlankLength);*

*LengthOfStringShow.Text:=FloattoStr(StringLength);*

*HowFarBall1UpPlankShow.Text:=FloattoStr(DistanceBall1StartsUpPlank);*

*UpdateModel;*

*end;*

## Code Appendix

### BallUnit

*unit BallUnit;*

*interface*

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,FMX.Objects;*

*type*

*TBall = Class(TImage)*

*Private*

*Public*

*Constructor Create(Stream:TStream; var AOwner);*

*Destructor Destroy;*

*End;*

*implementation*

*{$R \*.fmx}*

*Constructor TBall.Create(Stream: TStream; Var AOwner);*

*begin*

*Inherited Create(TComponent(AOwner));*

*Bitmap.LoadFromStream(Stream);*

*Parent := TFMXObject(AOwner);*

*end;*

*Destructor TBall.Destroy;*

*begin*

*Inherited Destroy;*

*end;*

*end.*

### MainMenuUnit

*unit MainMenuUnit;*

*interface*

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls, OwnModelUnit, PracticeUnit;*

*type*

*TMainMenuForm = class(TForm)*

*GoToOwnModelButton: TButton;*

*GoToPracticeButton: TButton;*

*WelcomeMessage: TLabel;*

*CloseProgramButton: TButton;*

*procedure GoToOwnModelButtonClick(Sender: TObject);*

*procedure GoToPracticeButtonClick(Sender: TObject);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*private*

*public*

*end;*

*var*

*MainMenuForm: TMainMenuForm;*

*implementation*

*{$R \*.fmx}*

*procedure TMainMenuForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*end;*

*procedure TMainMenuForm.GoToOwnModelButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Hide;*

*OwnModelForm.Show;*

*end;*

*procedure TMainMenuForm.GoToPracticeButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Hide;*

*PracticeForm.Show;*

*end;*

*end.*

### OwnModelUnit

*unit OwnModelUnit;*

*interface*

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,*

*FMX.Objects, Math, FMX.Edit,BallUnit;*

*type*

*TOwnModelForm = class(TForm)*

*Plank: TLine;*

*OwnModelFormWelcomeMessage: TLabel;*

*RequestForPlankAngle: TLabel;*

*RequestMassofBall1: TLabel;*

*RequestMassofBall2: TLabel;*

*RequestForLengthofPlank: TLabel;*

*RequestforDistanceBall1StartsupPlank: TLabel;*

*RequestforLengthofString: TLabel;*

*RequestforCoefficientOfFriction: TLabel;*

*AngleOfPlankSelector: TSpinBox;*

*AngleDegrees: TLabel;*

*MassOfBall1Selector: TEdit;*

*Ball1Kilograms: TLabel;*

*MassOfBall2Selector: TEdit;*

*LengthOfPlankSelector: TEdit;*

*LengthOfStringSelector: TEdit;*

*HowFarBall1UpPlankSelector: TEdit;*

*CoefficientOfFrictionSelector: TEdit;*

*Ball2Kilograms: TLabel;*

*PlankLengthMetres: TLabel;*

*StringLengthMetres: TLabel;*

*HowFarBall1UpPlankMetres: TLabel;*

*PlankString: TLine;*

*VerticalString: TLine;*

*AnimateModelButton: TButton;*

*UpdateModelButton: TButton;*

*ResetButton: TButton;*

*CloseProgramButton: TButton;*

*ReturnToMenuButton: TButton;*

*procedure AngleOfPlankSelectorChange(Sender: TObject);*

*procedure MassOfBall1SelectorChange(Sender: TObject);*

*procedure MassOfBall2SelectorChange(Sender: TObject);*

*procedure LengthOfPlankSelectorChange(Sender: TObject);*

*procedure LengthOfStringSelectorChange(Sender: TObject);*

*procedure HowFarBall1UpPlankSelectorChange(Sender: TObject);*

*procedure CoefficientOfFrictionSelectorChange(Sender: TObject);*

*procedure AnimatePlankAngle;*

*procedure AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*procedure AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX : real);*

*procedure AnimateBallsToCorrectPositions;*

*Procedure AnimateStringToCorrectHeight;*

*procedure UpdateModel;*

*procedure AnimateModelButtonClick(Sender: TObject);*

*procedure FormCreate(Sender: TObject);*

*procedure ResetToDefaultValues;*

*procedure UpdateModelButtonClick(Sender: TObject);*

*procedure CheckChosenValues;*

*procedure ResetButtonClick(Sender: TObject);*

*procedure FormKeyUp(Sender: TObject; var Key: Word; var KeyChar: Char;*

*Shift: TShiftState);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*procedure ReturnToMenuButtonClick(Sender: TObject);*

*procedure FormShow(Sender: TObject);*

*private*

*public*

*end;*

*var*

*OwnModelForm: TOwnModelForm;*

*PlankAngleDegrees, PlankAngleRadians, Ball1Mass, Ball2Mass, PlankLength, StringLength, DistanceBall1StartsUpPlank, CoefficientOfFriction, DefaultPlankAngleDegrees, DefaultBall1Mass, DefaultBall2Mass, DefaultPlankLength, DefaultStringLength, DefaultDistanceBall1StartsUpPlank, DefaultCoefficientOfFriction,HeightOfPlankPixels : real;*

*Ball1,Ball2: TBall;*

*CONST*

*g = 9.8;*

*SmoothnessConstant=100;*

*implementation*

*uses MainMenuUnit;*

*{$R \*.fmx}*

*function ConvertMetresToPixels(Input : real) : real;*

*begin*

*result:=Input\*(HeightOfPlankPixels/PlankLength);*

*end;*

*function CalculateHeightofPlankToUseInPixels : real;*

*begin*

*if OwnModelForm.Height < OwnModelForm.Width then result:= OwnModelForm.Height-150*

*else result:=OwnModelForm.Width-150;*

*end;*

*function CalculateSignificantFigures(Input:String) : integer;*

*var Stepper : integer;*

*begin*

*Result:=Length(Input);*

*for Stepper := 1 to Length(Input) do*

*begin*

*if Input[Stepper] = '.' then Result:=Length(Input) - 1*

*end;*

*end;*

*function RoundInputasString(Input:String;EndLength:Integer) : string;*

*var Stepper : integer;*

*Finished : boolean;*

*begin*

*if ord(Input[Endlength+1])>52*

*then begin*

*if ord(Input[Endlength])<57*

*then Input[EndLength]:=Char((Ord(Input[Endlength]))+1)*

*else begin*

*Input[Endlength]:='0';*

*Stepper:=EndLength-1;*

*while not Finished do*

*begin*

*if ord(input[Stepper])<57*

*then begin*

*Input[Stepper]:=Char((Ord(Input[Stepper]))+1);*

*Finished:=True;*

*end*

*else begin*

*Input[Stepper]:='0';*

*Stepper:=Stepper-1;*

*end;*

*end;*

*end;*

*end;*

*result:=Input;*

*end;*

*function MakeXDigits(Input:real;X:Integer) : real;*

*Var InputasString,InputasStringNoDecimal : String;*

*DecimalPlace : boolean;*

*Stepper,Counter,i,DecimalPosition : integer;*

*begin*

*InputasString:=FloattoStr(Input);*

*for Stepper := 1 to Length(InputAsString) do*

*begin*

*if InputasString[Stepper]='.'*

*then begin*

*DecimalPosition:=Stepper;*

*DecimalPlace:=true;*

*end*

*else begin*

*SetLength(InputAsStringNoDecimal,Length(InputAsStringNoDecimal)+1);*

*InputasStringNoDecimal[Length(InputAsStringNoDecimal)]:=InputasString[Stepper]*

*end;*

*end;*

*InputasStringNoDecimal:=RoundInputasString(InputasStringNoDecimal,X);*

*SetLength(InputasStringNoDecimal,X);*

*InputasString:=InputasStringNoDecimal;*

*if DecimalPlace=true*

*then begin*

*SetLength(InputasString,X+1);*

*for I := Length(InputasString) downto DecimalPosition+1 do*

*begin*

*InputasString[i]:=InputasString[i-1]*

*end;*

*InputasString[DecimalPosition]:='.';*

*end;*

*Result:=StrtoFloat(InputasString);*

*end;*

*function CalculateIfValidXDigitPositiveReal(Input:String; X:integer) : boolean;*

*begin*

*if (StrtoFloat(input)<0) or (CalculateSignificantFigures(Input)>X) then result:=false*

*else result := true;*

*end;*

*function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=OwnModelForm.Height-50-(HeightOfPlankPixels\*Cos((pi/2)-PlankAngleRadians));*

*end;*

*function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=70+(HeightOfPlankPixels\*Sin((pi/2)-PlankAngleRadians));*

*end;*

*procedure TOwnModelForm.AnimateBallsToCorrectPositions;*

*var NewBall1X,NewBall1Y,NewBall2X,NewBall2Y,BottomOfPlankX,BottomOfPlankY : real;*

*begin*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*NewBall1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*NewBall1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*NewBall2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*NewBall2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.AnimateFloat('position.X',NewBall1X,1);*

*Ball1.AnimateFloat('position.Y',NewBall1Y,1);*

*Ball2.AnimateFloat('position.X',NewBall2X,1);*

*Ball2.AnimateFloat('position.Y',NewBall2Y,1);*

*end;*

*procedure TOwnModelForm.AnimateModelButtonClick(Sender: TObject);*

*var UpSlope : string;*

*Counter : integer;*

*NewXBall1,NewYBall1,NewXBall2,NewYBall2,TimeForThisMovement,PartEndVelocity,Ball2YLocation,Ball1XLocation,Ball1YLocation,EachPartOfInitialMovementLength, InitialMovementLength,ExtraDisplacementLength,EachPartOfExtraDisplacementLength,Acceleration,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround,CurrentInitialSpeed,WaitingTime,LowestRatioForMovement,HighestRatioForMovement:real;*

*begin*

*CurrentInitialSpeed:=0;*

*WaitingTime:=0;*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope'*

*else UpSlope:= 'Stationery';*

*if UpSlope = 'UpSlope' then*

*begin*

*Acceleration:=((-Ball1Mass\*g\*((Sin(PlankAngleRadians))+(CoefficientofFriction\*(Cos(PlankAngleRadians)))))+(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 1 would fly off the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*CurrentInitialSpeed:=0;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:=Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength-(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end*

*else if UpSlope = 'DownSlope' then begin*

*Acceleration:=((Ball1Mass\*g\*((Sin(PlankAngleRadians))-(CoefficientofFriction\*(Cos(PlankAngleRadians)))))-(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if (PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength\*Sin(PlankAngleRadians)*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 2 would move higher than the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*CurrentInitialSpeed:=0;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-ExtraDisplacementLength+(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end*

*else if UpSlope = 'Stationery' then*

*begin*

*LowestRatioForMovement:= (CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians);*

*HighestRatioForMovement:=(-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians);*

*if HighestRatioForMovement<0*

*then Showmessage('The Ball would not move! For this angle and coefficient of friction, the system will only move where the Mass of Ball 2 divided by the Mass of Ball 1 is greater than ' + FloattoStr(MakeXDigits(LowestRatioForMovement,4)))*

*else Showmessage('The Ball would not move! For this angle and coefficient of friction, the system will only move where the Mass of Ball 2 divided by the Mass of Ball 1 is greater than ' + FloattoStr(MakeXDigits(LowestRatioForMovement,4)) + ' or smaller than ' + FloattoStr(MakeXDigits(HighestRatioForMovement,4)));*

*end;*

*end;*

*procedure TOwnModelForm.AnimatePlankAngle;*

*begin*

*Plank.AnimateFloat('rotationangle',90-PlankAngleDegrees,1);*

*end;*

*procedure TOwnModelForm.AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*begin*

*PlankString.AnimateFloat('rotationangle',90-PlankAngleDegrees,1);*

*PlankString.AnimateFloat('position.X',TopOfSlopeX+80,1);*

*PlankString.AnimateFloat('position.Y',TopOfSlopeY-30,1);*

*end;*

*procedure TOwnModelForm.AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);*

*begin*

*VerticalString.AnimateFloat('position.X',TopOfSlopeX+80,1);*

*VerticalString.AnimateFloat('position.Y',TopOfSlopeY-30,1);*

*end;*

*procedure TOwnModelForm.AnimateStringToCorrectHeight;*

*begin*

*PlankString.AnimateFloat('Height',ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank),1);*

*VerticalString.AnimateFloat('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank),1);*

*end;*

*procedure TOwnModelForm.UpdateModel;*

*var TopOfSlopeY, TopOfSlopeX : real;*

*begin*

*AnimatePlankAngle;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*AnimatePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX);*

*AnimateVerticalStringPosition(TopOfSlopeY,TopOfSlopeX);*

*AnimateBallsToCorrectPositions;*

*AnimateStringToCorrectHeight;*

*end;*

*procedure TOwnModelForm.UpdateModelButtonClick(Sender: TObject);*

*begin*

*CheckChosenValues;*

*UpdateModel;*

*end;*

*procedure TOwnModelForm.AngleOfPlankSelectorChange(Sender: TObject);*

*begin*

*if (StrtoInt(AngleofPlankSelector.Text)>89) or (StrtoInt(AngleofPlankSelector.Text)<1)then*

*begin*

*AngleOfPlankSelector.Text:=FloattoStr(DefaultPlankAngleDegrees);*

*ShowMessage('Not a valid angle of incline, please select an integer between 0 and 90');*

*end;*

*PlankAngleDegrees:=StrToInt(AngleofPlankSelector.Text);*

*PlankAngleRadians:=DegtoRad(PlankAngleDegrees);*

*end;*

*procedure TOwnModelForm.CoefficientOfFrictionSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(CoefficientOfFrictionSelector.Text,4) = false then*

*begin*

*CoefficientOfFrictionSelector.Text:=FloattoStr(DefaultCoefficientOfFriction);*

*ShowMessage('Not a valid coefficient of friction, please select a positive 4 digit or less real number');*

*end;*

*CoefficientOfFriction:=StrToFloat(CoefficientOfFrictionSelector.Text);*

*end;*

*procedure TOwnModelForm.ResetButtonClick(Sender: TObject);*

*begin*

*ResetToDefaultValues;*

*UpdateModel;*

*end;*

*procedure TOwnModelForm.ResetToDefaultValues;*

*begin*

*PlankLength:=DefaultPlankLength;*

*PlankAngleDegrees:= DefaultPlankAngleDegrees;*

*Ball1Mass:=DefaultBall1Mass;*

*Ball2Mass:=DefaultBall2Mass;*

*StringLength:=DefaultStringLength;*

*DistanceBall1StartsUpPlank:=DefaultDistanceBall1StartsUpPlank;*

*CoefficientOfFriction:=DefaultCoefficientOfFriction;*

*LengthOfPlankSelector.Text:=FloattoStr(DefaultPlankLength);*

*AngleOfPlankSelector.Text:= FloattoStr(DefaultPlankAngleDegrees);*

*MassOfBall1Selector.Text:=FloattoStr(DefaultBall1Mass);*

*MassOfBall2Selector.Text:=FloattoStr(DefaultBall2Mass);*

*LengthOfStringSelector.Text:=FloattoStr(DefaultStringLength);*

*HowFarBall1UpPlankSelector.Text:=FloattoStr(DefaultDistanceBall1StartsUpPlank);*

*CoefficientOfFrictionSelector.Text:=FloattoStr(DefaultCoefficientOfFriction);*

*end;*

*procedure TOwnModelForm.ReturnToMenuButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Show;*

*OwnModelForm.Hide;*

*end;*

*procedure TOwnModelForm.CheckChosenValues;*

*begin*

*if DistanceBall1StartsUpPlank+StringLength<PlankLength*

*then begin*

*ShowMessage('Not an accepeted change, this would result in the string not begin long enough to connect two particles in this plane. Resetting to Default Values.');*

*ResettoDefaultValues;*

*end;*

*if DistanceBall1StartsUpPlank>PlankLength*

*then begin*

*ShowMessage('Not an accepted change, the distance Ball 1 starts up the incline cannot be greater than the length of the plank. Resetting to Default Values.');*

*ResetToDefaultValues;*

*end;*

*if StringLength-PlankLength+DistanceBall1StartsUpPlank>PlankLength\*Sin(PlankAngleRadians)*

*then begin*

*ShowMessage('Not an accepted change, this would result in a slack string due to Ball 2 not being able to fall through the floor. Resetting to Default Values');*

*ResetToDefaultValues;*

*end;*

*procedure TOwnModelForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*Ball1.Destroy;*

*Ball2.Destroy;*

*end;*

*procedure TOwnModelForm.FormCreate(Sender: TObject);*

*var*

*Ballstream : TResourceStream;*

*begin*

*BallStream := TResourceStream.Create(MainInstance, 'BallImage', RT\_RCDATA);*

*Ball1:=TBall.Create(BallStream,Self);*

*Ball2:=TBall.Create(BallStream,Self);*

*DefaultPlankLength:=6;*

*DefaultPlankAngleDegrees:=30;*

*DefaultBall1Mass:=5;*

*DefaultBall2Mass:=5;*

*DefaultStringLength:=7;*

*DefaultDistanceBall1StartsUpPlank:=1;*

*DefaultCoefficientOfFriction:=0.3;*

*ResetToDefaultValues;*

*end;*

*procedure TOwnModelForm.FormKeyUp(Sender: TObject; var Key: Word; var KeyChar: Char; Shift: TShiftState);*

*begin*

*if Key=VKReturn then*

*begin*

*CheckChosenValues;*

*UpdateModel;*

*end;*

*end;*

*procedure TOwnModelForm.FormShow(Sender: TObject);*

*var TopOfSlopeY,TopOfSlopeX,BottomOfPlankY,BottomOfPlankX,Ball1X,Ball1Y,Ball2X,Ball2Y : real;*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*Plank.SetBounds(70,OwnModelForm.Height-50-HeightOfPlankPixels,50,HeightofPlankPixels);*

*ResetToDefaultValues;*

*Plank.RotationAngle:=90-PlankAngleDegrees;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*PlankString.rotationangle:=90-PlankAngleDegrees;*

*PlankString.Position.X:=TopOfSlopeX+80;*

*PlankString.Position.Y:=TopOfSlopeY-30;*

*VerticalString.Position.X:=TopOfSlopeX+80;*

*VerticalString.Position.Y:=TopOfSlopeY-30;*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*Ball1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*Ball1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*Ball2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*Ball2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.Position.X:=Ball1X;*

*Ball1.Position.Y:=Ball1Y;*

*Ball2.Position.X:=Ball2X;*

*Ball2.Position.Y:=Ball2Y;*

*PlankString.Height:=ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank);*

*VerticalString.Height:=ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank);*

*end;*

*procedure TOwnModelForm.HowFarBall1UpPlankSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(HowFarBall1UpPlankSelector.Text,3) = false then*

*begin*

*HowFarBall1UpPlankSelector.Text:=FloattoStr(DefaultDistanceBall1StartsUpPlank);*

*ShowMessage('Not a valid distance for ball 1 to start up plank, please select a positive 3 digit or less real number');*

*end;*

*DistanceBall1StartsUpPlank:=StrToFloat(HowFarBall1UpPlankSelector.Text);;*

*end;*

*procedure TOwnModelForm.LengthOfPlankSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(LengthOfPlankSelector.Text,3) = false then*

*begin*

*LengthOfPlankSelector.Text:=FloattoStr(DefaultPlankLength);*

*ShowMessage('Not a valid plank length, please select a positive 3 digit or less real number');*

*end;*

*PlankLength:=StrToFloat(LengthOfPlankSelector.Text);*

*end;*

*procedure TOwnModelForm.LengthOfStringSelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(LengthOfStringSelector.Text,3) = false then*

*begin*

*LengthOfStringSelector.Text:=FloattoStr(DefaultStringLength);*

*ShowMessage('Not a valid string length, please select a positive 3 digit or less real number');*

*end;*

*StringLength:=StrToFloat(LengthOfStringSelector.Text);*

*end;*

*procedure TOwnModelForm.MassOfBall1SelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(MassOfBall1Selector.Text,3) = false then*

*begin*

*MassOfBall1Selector.Text:=FloattoStr(DefaultBall1Mass);*

*ShowMessage('Not a valid mass of ball 1, please select a positive 3 digit or less real number');*

*end;*

*Ball1Mass:=StrToFloat(MassOfBall1Selector.Text);*

*end;*

*procedure TOwnModelForm.MassOfBall2SelectorChange(Sender: TObject);*

*begin*

*if CalculateIfValidXDigitPositiveReal(MassOfBall2Selector.Text,3) = false then*

*begin*

*MassOfBall2Selector.Text:=FloattoStr(DefaultBall2Mass);*

*ShowMessage('Not a valid mass of ball 2, please select a positive 3 digit or less real number');*

*end;*

*Ball2Mass:=StrToFloat(MassOfBall2Selector.Text);*

*end;*

*end.*

### PracticeUnit

*unit PracticeUnit;*

*interface*

*uses*

*System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,*

*FMX.Types, FMX.Graphics, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.StdCtrls,*

*FMX.Objects,Math,BallUnit, FMX.Edit;*

*type*

*TPracticeForm = class(TForm)*

*PracticeFormWelcomeMessage: TLabel;*

*Plank: TLine;*

*PlankString: TLine;*

*VerticalString: TLine;*

*TensionDisclaimer: TLabel;*

*AngleOfPlankDisclaimer: TLabel;*

*Ball1MassDisclaimer: TLabel;*

*Ball2MassDisclaimer: TLabel;*

*LengthOfPlankDisclaimer: TLabel;*

*LengthOfStringDisclaimer: TLabel;*

*HowFarBall1StartsUpPlankDisclaimer: TLabel;*

*CoefficientOfFrictionDisclaimer: TLabel;*

*AccelerationDisclaimer: TLabel;*

*TensionNewtons: TLabel;*

*AccelerationMetresPerSecondSquared: TLabel;*

*AngleOfPlankDegrees: TLabel;*

*Ball1MassKilograms: TLabel;*

*Ball2MassKilograms: TLabel;*

*LengthOfPlankMetres: TLabel;*

*LengthOfStringMetres: TLabel;*

*HowFarBall1UpPlankMetres: TLabel;*

*TensionShow: TLabel;*

*AccelerationShow: TLabel;*

*AngleOfPlankShow: TLabel;*

*Ball1MassShow: TLabel;*

*CoefficientOfFrictionShow: TLabel;*

*HowFarBall1UpPlankShow: TLabel;*

*LengthOfStringShow: TLabel;*

*LengthOfPlankShow: TLabel;*

*Ball2MassShow: TLabel;*

*QuestionsDisclaimer: TLabel;*

*Question1: TLabel;*

*Question2: TLabel;*

*Question3: TLabel;*

*CheckAndAnimateButton: TButton;*

*Question1Answer: TEdit;*

*Question2Answer: TEdit;*

*Question3Answer: TEdit;*

*NewProblemButton: TButton;*

*CloseProgramButton: TButton;*

*ReturnToMenuButton: TButton;*

*procedure FormResize(Sender: TObject);*

*procedure UpdateModel;*

*procedure ChangeBallsToCorrectPositions;*

*procedure ChangePlankAngle;*

*procedure ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX:Real);*

*procedure ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX:Real);*

*procedure ChangeStringToCorrectHeight;*

*procedure FormCreate(Sender: TObject);*

*procedure CheckAndAnimateButtonClick(Sender: TObject);*

*procedure NewProblemButtonClick(Sender: TObject);*

*procedure CreateProblem;*

*procedure FormShow(Sender: TObject);*

*procedure CloseProgramButtonClick(Sender: TObject);*

*procedure ReturnToMenuButtonClick(Sender: TObject);*

*private*

*public*

*end;*

*var*

*PracticeForm: TPracticeForm;*

*Acceleration,Tension, PlankAngleDegrees, PlankAngleRadians, Ball1Mass, Ball2Mass, PlankLength, StringLength, DistanceBall1StartsUpPlank, CoefficientOfFriction, HeightOfPlankPixels,InitialMovementLength,ExtraDisplacementLength : real;*

*UpSlope : String;*

*RandomInteger1,RandomInteger2 : integer;*

*Ball1,Ball2: TBall;*

*CONST*

*g = 9.8;*

*SmoothnessConstant=100;*

*implementation*

*uses MainMenuUnit;*

*{$R \*.fmx}*

*function ConvertMetresToPixels(Input : real) : real;*

*begin*

*result:=Input\*(HeightOfPlankPixels/PlankLength);*

*end;*

*function CalculateHeightofPlankToUseInPixels : real;*

*begin*

*if PracticeForm.Height < PracticeForm.Width then result:= PracticeForm.Height-150*

*else result:=PracticeForm.Width-150;*

*end;*

*function CalculateSignificantFigures(Input:String) : integer;*

*var Stepper : integer;*

*begin*

*Result:=Length(Input);*

*for Stepper := 1 to Length(Input) do*

*begin*

*if Input[Stepper] = '.' then Result:=Length(Input) - 1*

*end;*

*end;*

*function Random3DigitReal : real;*

*var FloatasString : String;*

*Counter,RandomDecisionValue : integer;*

*begin*

*for counter := 1 to 3 do*

*begin*

*FloatasString:=FloatasString+IntToStr(Random(9));*

*end;*

*RandomDecisionValue:=Random(100);*

*if RandomDecisionValue<33 then*

*begin*

*SetLength(FloatAsString,4);*

*FloatAsString[4]:=FloatAsString[3];*

*FloatAsString[3]:=FloatAsString[2];*

*FloatAsString[2]:='.';*

*end;*

*if (RandomDecisionValue>=33) and (RandomDecisionValue<66) then*

*begin*

*SetLength(FloatAsString,4);*

*FloatAsString[4]:=FloatAsString[3];*

*FloatAsString[3]:='.';*

*end;*

*result:=StrtoFloat(FloatasString);*

*end;*

*function RoundInputasString(Input:String;EndLength:Integer) : string;*

*var Stepper : integer;*

*Finished : boolean;*

*begin*

*if ord(Input[Endlength+1])>52*

*then begin*

*if ord(Input[Endlength])<57*

*then Input[EndLength]:=Char((Ord(Input[Endlength]))+1)*

*else begin*

*Input[Endlength]:='0';*

*Stepper:=EndLength-1;*

*while not Finished do*

*begin*

*if ord(input[Stepper])<57*

*then begin*

*Input[Stepper]:=Char((Ord(Input[Stepper]))+1);*

*Finished:=True;*

*end*

*else begin*

*Input[Stepper]:='0';*

*Stepper:=Stepper-1;*

*end;*

*end;*

*end;*

*end;*

*result:=Input;*

*end;*

*function MakeXDigits(Input:real;X:Integer) : real;*

*Var InputasString,InputasStringNoDecimal : String;*

*DecimalPlace : boolean;*

*Stepper,Counter,i,DecimalPosition : integer;*

*begin*

*InputasString:=FloattoStr(Input);*

*for Stepper := 1 to Length(InputAsString) do*

*begin*

*if InputasString[Stepper]='.'*

*then begin*

*DecimalPosition:=Stepper;*

*DecimalPlace:=true;*

*end*

*else begin*

*SetLength(InputAsStringNoDecimal,Length(InputAsStringNoDecimal)+1);*

*InputasStringNoDecimal[Length(InputAsStringNoDecimal)]:=InputasString[Stepper]*

*end;*

*end;*

*InputasStringNoDecimal:=RoundInputasString(InputasStringNoDecimal,X);*

*SetLength(InputasStringNoDecimal,X);*

*InputasString:=InputasStringNoDecimal;*

*if DecimalPlace=true*

*then begin*

*SetLength(InputasString,X+1);*

*for I := Length(InputasString) downto DecimalPosition+1 do*

*begin*

*InputasString[i]:=InputasString[i-1]*

*end;*

*InputasString[DecimalPosition]:='.';*

*end;*

*Result:=StrtoFloat(InputasString);*

*end;*

*function CalculateTension:real;*

*var Temp : real;*

*begin*

*if UpSlope='UpSlope' then Temp:=Ball2Mass\*(g-Acceleration);*

*if UpSlope='DownSlope' then Temp:=Ball2Mass\*(g+Acceleration);*

*Result:=MakeXDigits(Temp,4);*

*end;*

*procedure CreateRandomValues;*

*begin*

*PlankAngleDegrees:= RandomRange(0,90);*

*PlankAngleRadians:=DegtoRad(PlankAngleDegrees);*

*Ball1Mass:= Random3DigitReal;*

*Ball2Mass:= Random3DigitReal;*

*PlankLength:= Random3DigitReal;*

*DistanceBall1StartsUpPlank:= Random3DigitReal;*

*StringLength:= Random3DigitReal;*

*CoefficientOfFriction:= MakeXDigits(Random,4);*

*end;*

*function CalculateTopOfSlopeY (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=PracticeForm.Height-50-(HeightOfPlankPixels\*Cos((pi/2)-PlankAngleRadians));*

*end;*

*function CalculateTopOfSlopeX (HeightOfPlankPixels : real) : real;*

*begin*

*Result:=70+(HeightOfPlankPixels\*Sin((pi/2)-PlankAngleRadians));*

*end;*

*function SensibleM1Problem : boolean;*

*var TempAcceleration,InitialMovementLength,ExtraDisplacementLength,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround:real;*

*begin*

*if (DistanceBall1StartsUpPlank+StringLength<PlankLength)*

*or (DistanceBall1StartsUpPlank>PlankLength)*

*or (StringLength-PlankLength+DistanceBall1StartsUpPlank>PlankLength\*Sin(PlankAngleRadians))*

*then result:=false*

*else begin*

*result:=true;*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope'*

*else result:=false;*

*if UpSlope = 'UpSlope' then*

*begin*

*TempAcceleration:=((-Ball1Mass\*g\*((Sin(PlankAngleRadians))+(CoefficientofFriction\*(Cos(PlankAngleRadians)))))+(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*Acceleration:=MakeXDigits(TempAcceleration,7);*

*if Acceleration<0*

*then result:=false*

*else begin*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength then result:=false;*

*end;*

*end*

*else if UpSlope = 'DownSlope' then*

*begin*

*TempAcceleration:=((Ball1Mass\*g\*((Sin(PlankAngleRadians))-(CoefficientofFriction\*(Cos(PlankAngleRadians)))))-(Ball2Mass\*g))/(Ball1Mass+Ball2Mass);*

*Acceleration:=MakeXDigits(TempAcceleration,7);*

*if Acceleration<0*

*then result:=false*

*else begin*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*if PlankLength\*Sin(PlankAngleRadians)-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength > PlankLength\*Sin(PlankAngleRadians) then result:=false;*

*end;*

*end;*

*end;*

*end;*

*procedure TPracticeForm.ChangeBallsToCorrectPositions;*

*var NewBall1X,NewBall1Y,NewBall2X,NewBall2Y,BottomOfPlankX,BottomOfPlankY : real;*

*begin*

*BottomOfPlankY:=Plank.Position.Y+HeightOfPlankPixels-52;*

*BottomOfPlankX:=Plank.Position.X-18;*

*NewBall1X:=BottomOfPlankX+(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Cos(PlankAngleRadians));*

*NewBall1Y:=BottomOfPlankY-(ConvertMetresToPixels(DistanceBall1StartsUpPlank)\*Sin(PlankAngleRadians));*

*NewBall2X:=BottomOfPlankX+HeightOfPlankPixels\*Cos(PlankAngleRadians);*

*NewBall2Y:=BottomOfPlankY - ConvertMetresToPixels((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank);*

*Ball1.Position.X:=NewBall1X;*

*Ball1.Position.Y:=NewBall1Y;*

*Ball2.Position.X:=NewBall2X;*

*Ball2.Position.Y:=NewBall2Y;*

*end;*

*procedure TPracticeForm.ChangePlankAngle;*

*begin*

*Plank.RotationAngle:=90-PlankAngleDegrees;*

*end;*

*procedure TPracticeForm.ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX : real);*

*begin*

*PlankString.RotationAngle:=90-PlankAngleDegrees;*

*PlankString.Position.X:=TopOfSlopeX+6;*

*PlankString.Position.Y:=TopOfSlopeY-30;*

*end;*

*procedure TPracticeForm.ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX: Real);*

*begin*

*VerticalString.Position.X:=TopOfSlopeX+6;*

*VerticalString.Position.Y:=TopOfSlopeY-30;*

*end;*

*procedure TPracticeForm.CheckAndAnimateButtonClick(Sender: TObject);*

*var Counter : integer;*

*NewXBall1,NewYBall1,NewXBall2,NewYBall2,TimeForThisMovement,PartEndVelocity,Ball2YLocation,Ball1XLocation,Ball1YLocation,EachPartOfInitialMovementLength,EachPartOfExtraDisplacementLength,VelocityOfBall1WhenBall2HitGround,VelocityOFBall2WhenBall1HitGround,CurrentInitialSpeed,WaitingTime,LowestRatioForMovement,HighestRatioForMovement:real;*

*Question1Correct,Question2Correct,Question3Correct : boolean;*

*begin*

*if Ball2Mass>Ball1Mass\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='UpSlope'*

*else if Ball2Mass<Ball1Mass\*((-CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians)) then UpSlope:='DownSlope';*

*if UpSlope = 'UpSlope' then*

*begin*

*VelocityOfBall1WhenBall2HitGround:= Sqrt(2\*Acceleration\*((PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank));*

*ExtraDisplacementLength:=(((VelocityOfBall1WhenBall2HitGround)\*(VelocityOfBall1WhenBall2HitGround))/(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+(Sin(PlankAngleRadians)))));*

*InitialMovementLength:=(PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank;*

*end;*

*if UpSlope = 'DownSlope' then*

*begin*

*VelocityOfBall2WhenBall1HitGround:= Sqrt(2\*Acceleration\*DistanceBall1StartsUpPlank);*

*InitialMovementLength:=DistanceBall1StartsUpPlank;*

*ExtraDisplacementLength:=((VelocityOfBall2WhenBall1HitGround)\*(VelocityOfBall2WhenBall1HitGround))/(2\*g);*

*end;*

*if (UpSlope='UpSlope') and (Uppercase(Question1Answer.Text)='UP') then Question1Correct:=True;*

*if (UpSlope='DownSlope') and (Uppercase(Question1Answer.Text)='DOWN') then Question1Correct:=True;*

*if (RandomInteger1=1) and (Question2Answer.Text=FloatToStr(Tension)) then Question2Correct:=True;*

*if (RandomInteger1=2) and (Question2Answer.Text=FloatToStr(Acceleration)) then Question2Correct:=True;*

*if (RandomInteger1=3) and (Question2Answer.Text=FloatToStr(PlankAngleDegrees)) then Question2Correct:=True;*

*if (RandomInteger1=4) and (Question2Answer.Text=FloatToStr(Ball1Mass)) then Question2Correct:=True;*

*if (RandomInteger1=5) and (Question2Answer.Text=FloatToStr(Ball2Mass)) then Question2Correct:=True;*

*if (RandomInteger1=6) and (Question2Answer.Text=FloatToStr(CoefficientOfFriction)) then Question2Correct:=True;*

*if (RandomInteger2=1) and (Question3Answer.Text=FloatToStr(Tension)) then Question3Correct:=True;*

*if (RandomInteger2=2) and (Question3Answer.Text=FloatToStr(Acceleration)) then Question3Correct:=True;*

*if (RandomInteger2=3) and (Question3Answer.Text=FloatToStr(PlankAngleDegrees)) then Question3Correct:=True;*

*if (RandomInteger2=4) and (Question3Answer.Text=FloatToStr(Ball1Mass)) then Question3Correct:=True;*

*if (RandomInteger2=5) and (Question3Answer.Text=FloatToStr(Ball2Mass)) then Question3Correct:=True;*

*if (RandomInteger2=6) and (Question3Answer.Text=FloatToStr(CoefficientOfFriction)) then Question3Correct:=True;*

*if (RandomInteger2=7) and (Question3Answer.Text=FloatToStr(InitialMovementLength)) then Question3Correct:=True;*

*if (RandomInteger2=8) and (Question3Answer.Text=FloatToStr(ExtraDisplacementLength)) then Question3Correct:=True;*

*if (Question1Correct=True) and (Question2Correct=True) and (Question3Correct=True)*

*then begin*

*ShowMessage('Answers Correct. Animating!');*

*CurrentInitialSpeed:=0;*

*WaitingTime:=0;*

*if UpSlope = 'UpSlope' then*

*begin*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 1 would fly off the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank+(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball1XLocation:=NewXBall1;*

*Ball1YLocation:=NewYBall1;*

*CurrentInitialSpeed:=0;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*((CoefficientofFriction\*Cos(PlankAngleRadians))+Sin(PlankAngleRadians))\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewXBall1:=Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:=Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfExtraDisplacementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength-(Counter\*EachPartOfExtraDisplacementLength))),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end*

*else if UpSlope = 'DownSlope' then begin*

*EachPartOfInitialMovementLength:=InitialMovementLength/SmoothnessConstant;*

*EachPartOfExtraDisplacementLength:=ExtraDisplacementLength/SmoothnessConstant;*

*Ball2YLocation:=Ball2.Position.Y;*

*Ball1YLocation:=Ball1.Position.Y;*

*Ball1XLocation:=Ball1.Position.X;*

*if (PlankLength\*Sin(PlankAngleRadians))-StringLength+PlankLength-DistanceBall1StartsUpPlank+InitialMovementLength+ExtraDisplacementLength>PlankLength\*Sin(PlankAngleRadians)*

*then Showmessage('Chosen Values result in a problem beyond the M1 Specification, Ball 2 would move higher than the top of the incline')*

*else begin*

*for counter := 1 to SmoothnessConstant do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*Acceleration\*EachPartOfInitialMovementLength));*

*TimeForThisMovement:=(PartEndVelocity-CurrentInitialSpeed)/Acceleration;*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfInitialMovementLength);*

*NewXBall1:= Ball1XLocation - ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Cos(PlankAngleRadians));*

*NewYBall1:= Ball1YLocation + ConvertMetresToPixels((Counter\*EachPartOfInitialMovementLength)\*Sin(PlankAngleRadians));*

*Ball1.AnimateFloatDelay('position.Y',NewYBall1,TimeForThisMovement,WaitingTime);*

*Ball1.AnimateFloatDelay('position.X',NewXBall1,TimeForThisMovement,WaitingTime);*

*Ball2.AnimateFloatDelay('position.Y',NewYBall2,TimeForThisMovement,WaitingTime);*

*PlankString.AnimateFloatDelay('Height',ConvertMetresToPixels(PlankLength-(DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength))),TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-(Counter\*EachPartOfInitialMovementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeforThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(-2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation- ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*CurrentInitialSpeed:=0;*

*Ball2YLocation:=NewYBall2;*

*for counter := 1 to (SmoothnessConstant-1) do*

*begin*

*PartEndVelocity:=Sqrt((CurrentInitialSpeed\*CurrentInitialSpeed)+(2\*g\*EachPartOfExtraDisplacementLength));*

*TimeForThisMovement:=(2\*EachPartOfExtraDisplacementLength)/(CurrentInitialSpeed+PartEndVelocity);*

*NewYBall2:=Ball2YLocation+ConvertMetresToPixels(Counter\*EachPartOfExtraDisplacementLength);*

*Ball2.AnimateFloatDelay('position.y',NewYBall2,TimeForThisMovement,WaitingTime);*

*VerticalString.AnimateFloatDelay('Height',ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank-InitialMovementLength-ExtraDisplacementLength+(Counter\*EachPartOfExtraDisplacementLength)),TimeForThisMovement,WaitingTime);*

*WaitingTime:=WaitingTime+TimeForThisMovement;*

*CurrentInitialSpeed:=PartEndVelocity;*

*end;*

*end;*

*end;*

*end*

*else if (Question1Correct=False) and (Question2Correct=True) and (Question3Correct=True) then ShowMessage('Answer to Question 1 is incorrect')*

*else if (Question1Correct=True) and (Question2Correct=False) and (Question3Correct=True) then ShowMessage('Answer to Question 2 is incorrect')*

*else if (Question1Correct=True) and (Question2Correct=True) and (Question3Correct=False) then ShowMessage('Answer to Question 3 is incorrect')*

*else if (Question1Correct=False) and (Question2Correct=False) and (Question3Correct=True) then ShowMessage('Answers to Question 1 and Question 2 are incorrect')*

*else if (Question1Correct=False) and (Question2Correct=True) and (Question3Correct=False) then ShowMessage('Answers to Question 1 and Question 3 are incorrect')*

*else if (Question1Correct=True) and (Question2Correct=False) and (Question3Correct=False) then ShowMessage('Answers to Question 2 and Question 3 are incorrect')*

*else if (Question1Correct=False) and (Question2Correct=False) and (Question3Correct=False) then ShowMessage('Answers to all Questions are incorrect')*

*end;*

*procedure TPracticeForm.CloseProgramButtonClick(Sender: TObject);*

*begin*

*halt;*

*Ball1.Destroy;*

*Ball2.Destroy;*

*end;*

*procedure TPracticeForm.ChangeStringToCorrectHeight;*

*begin*

*PlankString.Height:=ConvertMetresToPixels(PlankLength-DistanceBall1StartsUpPlank);*

*VerticalString.Height:=ConvertMetresToPixels(StringLength-PlankLength+DistanceBall1StartsUpPlank);*

*end;*

*procedure TPracticeForm.UpdateModel;*

*var TopOfSlopeY,TopOfSlopeX : real;*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*ChangePlankAngle;*

*TopOfSlopeY:=CalculateTopOfSlopeY(HeightOfPlankPixels);*

*TopOfSlopeX:=CalculateTopOfSlopeX(HeightOfPlankPixels);*

*ChangePlankStringAngleAndPosition(TopOfSlopeY,TopOfSlopeX);*

*ChangeVerticalStringPosition(TopOfSlopeY,TopOfSlopeX);*

*ChangeBallsToCorrectPositions;*

*ChangeStringToCorrectHeight;*

*end;*

*procedure ResetValuestoZero;*

*begin*

*RandomInteger1:=0;*

*RandomInteger2:=0;*

*Tension:=0;*

*Acceleration:=0;*

*PlankAngleDegrees:=0;*

*Ball1Mass:=0;*

*Ball2Mass:=0;*

*PlankLength:=0;*

*StringLength:=0;*

*DistanceBall1StartsUpPlank:=0;*

*CoefficientOfFriction:=0;*

*end;*

*procedure TPracticeForm.CreateProblem;*

*Var QuestionsStream : TResourceStream;*

*QuestionsList : TStringList;*

*TensionQuestion,AccelerationQuestion,AngleQuestion,Ball1MassQuestion,Ball2MassQuestion,CoefficientQuestion : boolean;*

*begin*

*Question1Answer.Text:='';*

*Question2Answer.Text:='';*

*Question3Answer.Text:='';*

*ResetValuestoZero;*

*CreateRandomValues;*

*While not SensibleM1Problem = true do CreateRandomValues;*

*Tension:=CalculateTension;*

*QuestionsStream := TResourceStream.Create(MainInstance, 'Questions', RT\_RCDATA);*

*QuestionsList:=TStringList.Create;*

*QuestionsList.LoadFromStream(QuestionsStream);*

*while (RandomInteger1=RandomInteger2)*

*or ((RandomInteger1=6) And (RandomInteger2=4))*

*or ((RandomInteger1=4) And (RandomInteger2=6))*

*or ((RandomInteger1=6) And (RandomInteger2=3))*

*or ((RandomInteger1=3) And (RandomInteger2=6))*

*do begin*

*RandomInteger1:=RandomRange(1,6);*

*RandomInteger2:=RandomRange(1,8);*

*end;*

*case RandomInteger1 of*

*1 : TensionQuestion:=True;*

*2 : AccelerationQuestion:=True;*

*3 : AngleQuestion:=True;*

*4 : Ball1MassQuestion:=True;*

*5 : Ball2MassQuestion:=True;*

*6 : CoefficientQuestion:=True;*

*end;*

*case RandomInteger2 of*

*1 : TensionQuestion:=True;*

*2 : AccelerationQuestion:=True;*

*3 : AngleQuestion:=True;*

*4 : Ball1MassQuestion:=True;*

*5 : Ball2MassQuestion:=True;*

*6 : CoefficientQuestion:=True;*

*end;*

*Question1.Text:=QuestionsList[8];*

*Question2.Text:=QuestionsList[RandomInteger1-1];*

*Question3.Text:=QuestionsList[RandomInteger2-1];*

*if AccelerationQuestion = true then AccelerationShow.Text:='?'*

*else AccelerationShow.Text:=FloattoStr(Acceleration);*

*if TensionQuestion = true then TensionShow.Text:='?'*

*else TensionShow.Text:=FloattoStr(Tension);*

*if AngleQuestion = true then AngleofPlankShow.Text:='?'*

*else AngleOfPlankShow.Text:=FloattoStr(PlankAngleDegrees);*

*if Ball1MassQuestion = true then Ball1MassShow.Text:='?'*

*else Ball1MassShow.Text:=FloattoStr(Ball1Mass);*

*if Ball2MassQuestion = true then Ball2MassShow.Text:='?'*

*else Ball2MassShow.Text:=FloattoStr(Ball2Mass);*

*if CoefficientQuestion = true then CoefficientOfFrictionShow.Text:='?'*

*else CoefficientOfFrictionShow.Text:=FloattoStr(CoefficientOfFriction);*

*LengthOfPlankShow.Text:=FloattoStr(PlankLength);*

*LengthOfStringShow.Text:=FloattoStr(StringLength);*

*HowFarBall1UpPlankShow.Text:=FloattoStr(DistanceBall1StartsUpPlank);*

*UpdateModel;*

*end;*

*procedure TPracticeForm.FormCreate(Sender: TObject);*

*var BallStream:TResourceStream;*

*begin*

*BallStream := TResourceStream.Create(MainInstance, 'BallImage', RT\_RCDATA);*

*Ball1:=TBall.Create(BallStream,Self);*

*Ball2:=TBall.Create(BallStream,Self);*

*Randomize;*

*end;*

*procedure TPracticeForm.FormShow(Sender: TObject);*

*begin*

*HeightofPlankPixels:=CalculateHeightofPlanktoUseInPixels;*

*Plank.SetBounds(70,PracticeForm.Height-50-HeightOfPlankPixels,50,HeightofPlankPixels);*

*CreateProblem;*

*end;*

*procedure TPracticeForm.NewProblemButtonClick(Sender: TObject);*

*begin*

*CreateProblem;*

*end;*

*procedure TPracticeForm.ReturnToMenuButtonClick(Sender: TObject);*

*begin*

*MainMenuForm.Show;*

*PracticeForm.Hide;*

*end;*

*end.*