```
/*****************************************************************************
*   TITLE: Space Simulator
*
*   LAST MODIFIED: 15th of May 2011, 18:09 PM, +2 GMT
*
*   AUTHOR: Johan Näslund
*
*   DESCRIPTION: This application is a simulator of the universe and the
*   gravitational laws within it. It manages a space containing different
*   space objects and calculates the gravity between them to simulate a
*   universe.
*
*   The application responds to some user input (see the
*   "Space Simulator Overview.docx" file) and displays simple graphics
*   drawn with OpenGL - GLUT.
*
*   REQUIREMENTS:
*   •   The code was written entirely in the Code::Blocks IDE. To compile the
*   code glut.h, libglut32.a and glut32.dll has to be added to the IDE and
*   system folders if these aren't included already. These files can be found
*   in the 'dependencies' folder, together with some simple instructions.
*   •   The code has only been tested to compile on the Windows Vista
*   Operating System. It will most likely also run on all other Windows
*   Operating systems released after Windows 95. On the earlier versions
*   support for OpenGL has to be installed manually.
*
*   NOTES:
*   •   At the time of the creation of this program I had no experience
*   with GLUT before and very little experience with OpenGL. Therefore the
*   code concerning GLUT and OpenGL will probably not be optimal. It was
*   simply a means to an end of presenting my results in a more concrete way.
*   •   All the lines within the code have been kept at 78 characters long
*   for the sake of proper printing line length.
*   •   The coding standard used in this project is inspired by:
*       http://www.possibility.com/Cpp/CppCodingStandard.html
*
*   TODO:
*   •   Have the radius of the objects in space scale with the map scale.
*   (However at the moment that isn't of a very high priority and as such it
    has been left on the todo list.)
*
*****************************************************************************/

/*****************************************************************************
*   FILE: main.cpp
*
*   FUNCTION: The file containing the main function.
*
*   PURPOSE: The reason for keeping the main function in a separate file is to
*   keep an easy to understand structure of the code, anyone can easily find
*   and edit the code that is executed at run-time.
*
*****************************************************************************/

#include "Draw.h"
#include "Space.h"

/**
    Name: main(int, char*)
    Function: Contains the code to be executed at run-time.
**/
int main(int argc, char* argv[]){
    //  create and initialize window
    Window window("Space Simulator", 800, 800, argc, argv);
    window.Initialize();

    /*  START: Create the universe   */
    //  construct a space
    //  the argument given is how many seconds should pass per update,
    //  less seconds per update yields more accurate simulation
    Space space(150);
    /*  START: Declare objects  */
    Star* sun =         new Star("Sun", //name
                                 1.9891e30, //mass
                                 0.025,   //radius
                                 Coordinate(0, 0), //position
                                 Coordinate(0, 0), //velocity
                                 1.00, 1.00, 0.00); //rgb colour
    Planet* mercury =   new Planet("Mercury", //name
                                   6.083e10, //mass
                                   0.0125, //radius
                                   Coordinate(5790906e4, 0), //position
                                   Coordinate(0, 47870), //velocity
                                   0.6, 0.6, 0.6); //rgb colour
    Planet* venus =     new Planet("Venus", 4.8685e24, 0.0125,
                                   Coordinate(1082089e5, 0),
                                   Coordinate(0, 35020),
                                   1.0, 0.5, 0.5);
    Planet* earth =     new Planet("Earth", 5.9736e24, 0.0125,
                                   Coordinate(149598261e3,  0),
                                   Coordinate(0, 29783),
```

```
                                    0, 1.0, 0);
        Planet* mars =      new Planet("Mars", 4.185e23, 0.0125 ,
                                    Coordinate(227939100e3, 0),
                                    Coordinate(0, 24077),
                                    1.0, 0, 0);
        Planet* jupiter =   new Planet("Jupiter",1.8986e27, 0.0125,
                                    Coordinate(778547200e3, 0),
                                    Coordinate(0, 13.07e3),
                                    0.8 ,0.4 ,0);
        Planet* saturn =    new Planet("Saturn",8.2713e14, 0.0125,
                                    Coordinate(1433449370e3, 0),
                                    Coordinate(0, 9.69e3),
                                    0.7, 0.5, 0);
        Planet* uranus =    new Planet("Uranus",8.6810e25, 0.0125,
                                    Coordinate(2876679082e3, 0),
                                    Coordinate(0, 6.81e3),
                                    0, 0, 0.8);
        Planet* neptune =   new Planet("Neptune", 1.0243e26, 0.0125,
                                    Coordinate(4452940833e3, 0),
                                    Coordinate(0, 5.43e3),
                                    0, 0, 1.0);
        Moon* moon =        new Moon(earth, //"owner" planet
                                "Moon", //name
                                7.3477e22,  //mass
                                0.00625,    //radius
                                384399e3, //distance from owner
                                0.8, 0.8, 0.8); //rgb colour
        Moon* mJupiter =    new Moon(jupiter, "Moon", 7.3477e22,
                                0.00625, 184399e4, 0.8, 0.8, 0.8);

        /*  END: Declaration of objects */
        /*  START: Add objects to space */
        space.AddObjectToSpace(sun);
        space.AddObjectToSpace(mercury);
        space.AddObjectToSpace(venus);
        space.AddObjectToSpace(earth);
        space.AddObjectToSpace(moon);
        space.AddObjectToSpace(mars);
        space.AddObjectToSpace(jupiter);
        space.AddObjectToSpace(mJupiter);
        space.AddObjectToSpace(saturn);
        space.AddObjectToSpace(uranus);
        space.AddObjectToSpace(neptune);
        /*  END: Add objects to space   */
        /*  END: Create the universe */
        /*  START: Construct the draw object    */
        //  give the draw class object pointers to
        //  the window to draw in and the space to draw
        Draw * draw = new Draw(&window, &space, 10);
        //  sets the instance to this draw
        draw->SetInstance(draw);
        /*  END: Construct the draw object  */
        //  start drawing
        draw->Start();
        return 0;
    }

```

```
/*****************************************************************************
*   FILE: Window.h
*
*   FUNCTION: The window class creates and initializes a window.
*
*   PURPOSE: Having a separate window class means it will be easier to handle
*   windows.
*
*****************************************************************************/

#ifndef _Window_
#define _Window_

#include <string>

class Window{
    public:
    /** Constructors    **/
    //  constructs a window with a name char[], width and height ints,
    //  and forwards the int and char** main arguments
    Window(const char[], int, int, int, char**);
    /** Member Functions   **/
    //  initializes a window with the arguments given at construction
    void            Initialize();
    /** Getters and Setters **/
    int             GetWidth()
                        {return mWidth;}
    void            SetWidth(int width)
                        {mWidth = width;}
    int             GetHeight()
                        {return mHeight;}
    void            SetHeight(int height)
                        {mHeight = height;}

    private:
    /** Class Members   **/
    const char**    mppName;
    int             mWidth;
    int             mHeight;
    int             mMainArgC;
    char **         mppMainArgV;
};
#endif

```

```cpp
/*****************************************************************************
*   FILE: Window.cpp
*
*   FUNCTION: The window class creates and initializes a window.
*
*   PURPOSE: Having a separate window class means it will be easier to handle
*   windows.
*
*****************************************************************************/

#include "Window.h"
#include <GL/glut.h>
#include <windows.h>

/*****************************************************************************
 * Constructors
 *
 *****************************************************************************/

/**
    Name: Window(const char[], int, int, int, char**)
    Function: Constructs a window with a name char[], width and height ints,
    and forwards the int and char** main arguments
**/
Window::Window(const char pName[], int width, int height,
               int mainArgC, char* pMainArgV[]){
    mppName = &pName;
    mWidth = width;
    mHeight = height;

    mMainArgC = mainArgC;
    mppMainArgV = pMainArgV;
}

/*****************************************************************************
* Class methods
*
*****************************************************************************/

/**
    Name: Initialize()
    Function: Initializes the window by setting its display mode, size and
    title. Also forwards any arguments sent to the main function to GLUT(if
    constructed correctly).
**/
void Window::Initialize(){
    //  sends main arguments to GLUT
    glutInit(&mMainArgC, mppMainArgV);
    //  set the window size
    glutInitWindowSize (mWidth, mHeight);
    //  sets the display mode to:
    //  double buffers, red green blue colouring, and allows depth buffering
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    //  create a window with the name argument as title
    glutCreateWindow (*mppName);
}

```

```
/*****************************************************************************
*   FILE: Draw.h
*
*   FUNCTION: This class is responsible for ALL drawing done in the application.
*   It uses a window as the target for drawing and a space for the content to
*   draw. The class manages GLUT fucntion calls, variables, user-input,
*   scaling from window coordinates to real coordinates and vice versa.
*
*   PURPOSE: By collecting all the drawing done in the application in one
*   class it keeps the cohesion at a manageable level. It might also make it
*   easier to draw in multiple windows later on if it will ever be supported.
*
*****************************************************************************/

#ifndef _Draw_
#define _Draw_

#include "Window.h"
#include "Space.h"
#include <GL/glut.h>
#include <windows.h>

/* Solution for encapsulating GLUT inspired by:
    http://paulsolt.com/2009/07/openglglut-classes-oop-and-problems/ */

class Draw{
    public:
    /** Constructors    **/
    //  constructs a draw object with a target window to draw in,
    //  a space to draw and a double starting scale
    Draw(Window*, Space*, double);

    /** Member Functions   **/
    //  starts the GLUT main loop
    void            Start();
    //  draws a sphere at a coordinate with a double size
    void            DrawSphere(Coordinate, double);
    //  scale a meters double to screen percentage
    double          ToScale(double);
    //  scale a screen percentage double to meters
    double          FromScale(double);
    // draw all the stars in space
    void            DrawStars();
    //  draw all the planets in space
    void            DrawPlanets();
    //  draw all the moons in space
    void            DrawMoons();
    //  draws lighting on the argument star
    void            DrawLighting(Star*);
    //  draws lighting on the argument planet
    void            DrawLighting(Planet*);
    //  draws lighting on the argument moon
    void            DrawLighting(Moon*);

    /** Functions called by GLUT  **/
    //  calls my own non-static display handler
    static void     DisplayWrapper()
                        {mspInstance->Display();}
    //  calls my own non-static idle handler
    static void     IdleWrapper()
                        {mspInstance->Idle();};
    //  calls my own non-static keyboard-input handler
    static void     KeyboardWrapper(unsigned char key, int x, int y)
                        {mspInstance->Keyboard(key, x, y);}
    //  calls my own non-static mouse-input handler
    static void     MouseWrapper(int button, int state, int x, int y)
                        {mspInstance->Mouse(button, state, x, y);}

    /** My own non-static GLUT functions **/
    //  displays everything in the space
    void            Display();
    //  the function called when GLUT is idle
    void            Idle();
    //  the function called when an ASCII key is pressed
    void            Keyboard(unsigned char, int, int);
    //  the function called when a mouse button is clicked
    void            Mouse(int, int, int, int);

    /** Getters and Setters **/
    static void     SetInstance(Draw * instance);
    void            SetScale(double scaleAu){mScaleAu = scaleAu;}
    double          GetScale(){return mScaleAu;}
    double          CalculateNewObjectSpeed(int, int);

    private:
    /** Class members   **/
    static  Draw*                       mspInstance;
    Window*                             mpWindow;
    Space*                              mpSpace;
    double                              mScaleAu;
    std::list<SpaceObject*>             mLookAt;
```

```
92        std::list<SpaceObject*>::iterator   mLookAtIterator;
93
94    };
95
96    #endif
97
```

```cpp
/*****************************************************************************
*   FILE: Draw.cpp
*
*   FUNCTION: This class is responsible for ALL drawing done in the application.
*   It uses a window as the target for drawing and a space for the content to
*   draw. The class manages GLUT fucntion calls, variables, user-input,
*   scaling from window coordinates to real coordinates and vice versa.
*
*   PURPOSE: By collecting all the drawing done in the application in one
*   class it keeps the cohesion at a manageable level. It might also make it
*   easier to draw in multiple windows later on if it will ever be supported.
*
*****************************************************************************/

#include "Draw.h"

/*****************************************************************************
 * Constructors
 *
 *****************************************************************************/

/**
    Name: Draw(Window*, Space*, double)
    Function: Constructs a draw object by giving it a window to draw in, a
    space to draw as well as the starting scale of the window. This function
    also sets up the material and light variables to be used when drawing as
    well specifying the functions to be called by GLUT.
**/
Draw::Draw(Window* pWindow, Space* pSpace, double scaleAu)
{
    mpWindow        = pWindow;
    mpSpace         = pSpace;
    mScaleAu        = scaleAu;
    mLookAt         = mpSpace->GetObjectsInSpace();
    mLookAtIterator = mLookAt.begin();

    //  enable lighting
    const GLfloat lightAmbient[]  = {0.0, 0.0, 0.0, 1.0};
    const GLfloat lightDiffuse[]  = {1.0, 1.0, 1.0, 1.0};

    //  enable materials
    glEnable(GL_COLOR_MATERIAL);
    //  enable lighting
    glEnable(GL_LIGHTING);

    /*  START: Enable light emission from all suns   */
    //  create a copy of the list of the stars in space
    std::list<Star*> starsList = mpSpace->GetStarsInSpace();
    //  create a star list iterator
    std::list<Star*>::iterator starsIterator;
    //  iterate through the stars in space
    for(starsIterator = starsList.begin();
        starsIterator != starsList.end(); starsIterator++)
    {
        //  set a temporary emitter pointer
        Star* pStar = *starsIterator;
        if(pStar->GetLightSource())
        {
            //  enable lighting
            glEnable(pStar->GetLightSource());
            //  specify ambient lighting
            glLightfv(pStar->GetLightSource(), GL_AMBIENT,  lightAmbient);
            //  specify diffuse lighting
            glLightfv(pStar->GetLightSource(), GL_DIFFUSE,  lightDiffuse);
        }
    }
    /*  END: Enable lighting from all suns  */

    /*  START: Set materials    */
    const GLfloat matAmbient[]    = {0.7, 0.7, 0.7, 1.0};
    const GLfloat matDiffuse[]    = {0.8, 0.8, 0.8, 1.0};
    const GLfloat highShininess[] = {50.0};

    //  specify material ambient properties
    glMaterialfv(GL_FRONT, GL_AMBIENT,   matAmbient);
    //  specify diffuse properties
    glMaterialfv(GL_FRONT, GL_DIFFUSE,   matDiffuse);
    //  specify shininess
    glMaterialfv(GL_FRONT, GL_SHININESS, highShininess);
    /*  END: Set materials  */
    //  point to my static GLUT handlers
    //  which in turn will call for my non-static handlers
    glutDisplayFunc(DisplayWrapper);
    glutIdleFunc(IdleWrapper);
    glutKeyboardFunc(KeyboardWrapper);
    glutMouseFunc(MouseWrapper);
}

/*****************************************************************************
* Member Functions
*
```

```
92  ********************************************************************************/
93
94  /**
95      Name: Start()
96      Function: Starts the GLUT main loop.
97  **/
98  void Draw::Start()
99  {
100     glutMainLoop();
101 }
102
103 /**
104     Name: DrawSphere(Coordinate, double)
105     Function: Draws a sphere using the arguments radius(in screen percentage)
106     and a position coordinate(that is being scaled to screen percentages
107     within the function).
108 **/
109 void Draw::DrawSphere(Coordinate position, double radius)
110 {
111     //  start new state
112     glPushMatrix();
113     //  set the position of the object
114     glTranslated(ToScale(position.GetX()), ToScale(position.GetY()), 0.0);
115     //  draw object
116     glutSolidSphere(radius, 80, 80);
117     //  go back to previous state
118     glPopMatrix();
119 }
120
121 /**
122     Name: DrawLighting(Star*)
123     Function: Draws lighting on the argument star.
124 **/
125 void Draw::DrawLighting(Star* pStar)
126 {
127             //  draws light positioned at the center of the star
128             float gLightPosition[] = {0.0, 0.0, 1.0, 1.0};
129             glLightfv(GL_LIGHT0, GL_POSITION, gLightPosition);
130 }
131
132 /**
133     Name: DrawLighting(Planet*)
134     Function: Draws lighting from all the stars in space on the argument
135     planet.
136 **/
137 void Draw::DrawLighting(Planet* pPlanet)
138 {
139     //  create a copy of the list of the stars in space
140     std::list<Star*> starsList = mpSpace->GetStarsInSpace();
141     //  create a star list iterator
142     std::list<Star*>::iterator starsIterator;
143     //  iterate through the stars in space
144     for(starsIterator = starsList.begin();
145         starsIterator != starsList.end(); starsIterator++)
146     {
147         //  set a temporary emitter pointer
148         Star* pStar = *starsIterator;
149         //  calculate difference in x-axis
150         double x
151                = pStar->GetPosition().GetX() - pPlanet->GetPosition().GetX();
152         //  calculate difference in y-axis
153         double y
154                = pStar->GetPosition().GetY() - pPlanet->GetPosition().GetY();
155         //  set the light position towards the star
156         float gLightPosition[] = {x, y, 1.0, 1.0};
157         glLightfv(pStar->GetLightSource(), GL_POSITION, gLightPosition);
158     }
159 }
160
161 /**
162     Name: DrawLighting(Moon*)
163     Function: Draws lighting from all the stars in space on the argument
164     moon.
165 **/
166 void Draw::DrawLighting(Moon* pMoon)
167 {
168     //  create a copy of the list of the stars in space
169     std::list<Star*> starsList = mpSpace->GetStarsInSpace();
170     //  create a star list iterator
171     std::list<Star*>::iterator starsIterator;
172     //  iterate through the stars in space
173     for(starsIterator = starsList.begin();
174         starsIterator != starsList.end(); starsIterator++)
175     {
176             //  set a temporary emitter pointer
177             Star* pStar = *starsIterator;
178             //  calculate difference in x-axis
179             float x =
180                pStar->GetPosition().GetX() - pMoon->GetPosition().GetX();
181             //  calculate difference in y-axis
182             float y =
```

```
                pStar->GetPosition().GetY() - pMoon->GetPosition().GetY();
            //  set the light position towards the star
            float gLightPosition[] = {x, y, 1.0, 1.0};
            glLightfv(pStar->GetLightSource(), GL_POSITION, gLightPosition);
        }
    }

    /**
        Name: DrawStars()
        Function: Draws all the stars in space.
    **/
    void Draw::DrawStars()
    {
        //  save a copy of the stars list
        std::list<Star*> starsList = mpSpace->GetStarsInSpace();
        //  iterate through the list
        for(std::list<Star*>::iterator starsIterator = starsList.begin();
            starsIterator != starsList.end(); starsIterator++)
        {
            //  create a temporary pointer to the current object
            Star* pStar = *starsIterator;
            //  set the colour to be used
            glColor3f(pStar->GetRed(),
                      pStar->GetGreen(),
                      pStar->GetBlue());
            //  draw the lighting on the current object
            DrawLighting(pStar);
            //  draw a sphere of the object
            DrawSphere(pStar->GetPosition(), pStar->GetRadius());
        }

    }

    /**
        Name: DrawPlanets()
        Function: Draws all the planets in space.
    **/
    void Draw::DrawPlanets()
    {
        //  save a copy of the planets list
        std::list<Planet*> planetsList = mpSpace->GetPlanetsInSpace();
        //  iterate through the list
        for(std::list<Planet*>::iterator planetsIterator = planetsList.begin();
            planetsIterator != planetsList.end(); planetsIterator++)
        {
            //  create a temporary pointer to the current object
            Planet * pPlanet = *planetsIterator;
            //  set the colours to be used
            glColor3f(pPlanet->GetRed(),
                      pPlanet->GetGreen(),
                      pPlanet->GetBlue());
            //  draw the lighting on the current object
            DrawLighting(pPlanet);
            //  draw a sphere of the current object
            DrawSphere(pPlanet->GetPosition(), pPlanet->GetRadius());
        }
    }

    /**
        Name: DrawMoons()
        Function: Draws all the moons in space.
    **/
    void Draw::DrawMoons()
    {
        //  save a copy of the moons list
        std::list<Moon*> moonsList = mpSpace->GetMoonsInSpace();
        //  iterate through the list
        for(std::list<Moon*>::iterator moonsIt = moonsList.begin();
            moonsIt != moonsList.end(); moonsIt++)
        {
            //  create a temporary pointer to the current object
            Moon * pMoon = *moonsIt;
            //  set the colour to be used
            glColor3f(pMoon->GetRed(),
                      pMoon->GetGreen(),
                      pMoon->GetBlue());
            //  draw the lighting on the current object
            DrawLighting(pMoon);
            //  draw a sphere of the current object
            DrawSphere(pMoon->GetPosition(), pMoon->GetRadius());
        }
    }

    /**
        Name: ToScale(double)
        Function: Scales the argument from meters to window percentages.
    **/
    double Draw::ToScale(double meters)
    {
        //  One Astronomical Unit(roughly the distance between the sun and the earth)
        //  is equal to 149598e6 meters
```

```
274         return (meters/(149598e6*mScaleAu));
275     }
276 
277     /**
278         Name: FromScale(double)
279         Function: Scales the argument from window percentages to meters.
280     **/
281     double Draw::FromScale(double screenPercentage)
282     {
283         return screenPercentage*149598e6*mScaleAu;
284     }
285 
286     /**
287         Name: CalculateNewObjectSpeed(int, int)
288         Function: Calculates object speed on one axis, the speed being relative
289         to the distance between the two int arguments(pixels) and the current
290         scale.
291     **/
292     double Draw::CalculateNewObjectSpeed(int buttonDown, int buttonUp)
293     {
294         return mScaleAu*15*(buttonUp - buttonDown);
295     }
296 
297     /*****************************************************************************
298     * GLUT handlers
299     *
300     *****************************************************************************/
301 
302     /**
303         Name: Idle()
304         Function: The function called when GLUT is idle.
305     **/
306     void Draw::Idle()
307     {
308         for(int i = 0; i < 100; i++)
309         {
310             mpSpace->CalculateGravity();
311             mpSpace->PassTime();
312         }
313         //  set the matrix to default
314         glLoadIdentity();
315         SpaceObject * pFocus = *mLookAtIterator;
316         gluLookAt(
317                     //  the position of the eye
318                     ToScale(pFocus->GetPosition().GetX()),
319                     ToScale(pFocus->GetPosition().GetY()), 1.0,
320                     //  the position of the object
321                     ToScale(pFocus->GetPosition().GetX()),
322                     ToScale(pFocus->GetPosition().GetY()), 0.0,
323                     //  the angular rotation around the x, y, and x axises
324                     0, 1.0, 0);
325         //  call the display function
326         Display();
327         //  put the application to sleep for 1 ms
328         Sleep(1);
329     }
330 
331     /**
332         Name: Display()
333         Function: Displays all objects in space.
334     **/
335     void Draw::Display()
336     {
337         //  clear the window
338         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
339 
340         DrawStars();
341         DrawPlanets();
342         DrawMoons();
343         //  swap the back and front buffer
344         glutSwapBuffers();
345     }
346 
347     /**
348         Name: Keyboard(unsigned char, int, int)
349         Function: The function for handling keyboard input. It takes in three
350         arguments: The key pressed(ASCII) as well as the two mouse coordinates(pixels) when
    the key
351         is pressed.
352     **/
353     void Draw::Keyboard(unsigned char key, int x, int y)
354     {
355 
356         switch (key)
357         {
358             /* Exits the application   */
359             case 'q':
360                 exit(0);
361                 break;
362             /* Zooms in    */
363             case '+':
```

```
364                 if(mScaleAu > 0.2)
365                 {
366                     if(mScaleAu < 5)
367                     {
368                         mScaleAu -= 0.2;
369                     }
370                     else if(mScaleAu < 1)
371                     {
372                         mScaleAu -= 0.01;
373                     }
374                     else if(mScaleAu >= 5)
375                     {
376                         mScaleAu--;
377                     }
378                 }
379                 else
380                 {
381                     //zoom in is at max
382                 }
383                 break;
384             /* Zooms out   */
385             case '-':
386                 if(mScaleAu < 30)
387                 {
388                     if(mScaleAu < 5)
389                     {
390                         mScaleAu += 0.2;
391                     }
392                     else if(mScaleAu < 1)
393                     {
394                         mScaleAu += 0.2;
395                     }
396                     else
397                     {
398                         mScaleAu++;
399                     }
400                 }
401                 else
402                 {
403                     //zoom out is at max
404                 }
405                 break;
406             /*  Look at next object in space */
407             case 'n':
408                 //  if the end of the list isnt reached
409                 if(mLookAtIterator != mLookAt.end())
410                 {
411                     mLookAtIterator++;
412                 }
413                 //  if the end of the list is reached
414                 else
415                 {
416                     //  iterate from first object
417                     mLookAtIterator = mLookAt.begin();
418                 }
419                 break;
420             /*  Delete last object in space */
421             case 127:
422                 //  if looking at planet to be deleted
423                 if(*mLookAtIterator == mpSpace->GetObjectsInSpace().back())
424                 {
425                     //  start looking at firstobject
426                     mLookAtIterator = mLookAt.begin();
427                 }
428                 //  point to the last star in space
429                 Star* pStar = mpSpace->GetStarsInSpace().back();
430                 //  disable the last star in space's light emission
431                 glDisable(pStar->GetLightSource());
432                 //  deletes the last inserted object in space
433                 mpSpace->PopObjectFromSpace();
434                 //  if there are any stars in space
435                 if(!mpSpace->GetStarsInSpace().empty())
436                 {
437                     //  point to the last star in space
438                     pStar = mpSpace->GetStarsInSpace().back();
439                     //  enable the last star in space's light emission
440                     glEnable(pStar->GetLightSource());
441                 }
442                 // save the new list of objects
443                 mLookAt = mpSpace->GetObjectsInSpace();
444                 break;
445         }
446     }
447
448     int gPosX = 0;
449     int gPosY = 0;
450
451     /**
452         Name: Mouse(int, int, int, int)
453         Function: The function for handling mouse input. Takes in an argument for
454         which button pressed, the button press state(up or down) as well as
```

```
455         coordinates(pixels) for the button click.
456     **/
457     void Draw::Mouse(int button, int state, int x, int y){
458             switch(button)
459             {
460                 /*  This case creates a planet at the position of the mouse.
461                     It also calculates the speed of the object by using
462                     the button release coordinates. */
463                 case GLUT_LEFT_BUTTON:
464                 if(state == GLUT_DOWN){
465                     //   rememebers the button down coordinates
466                     gPosX = x;
467                     //   rememebers the button down coordinates
468                     gPosY = y;
469                 }else if(state == GLUT_UP){
470                     //  create a coordinate system using half window width
471                     double hWidth = mpWindow->GetWidth()/2;
472                     //  create a coordinate system using half window height
473                     double hHeight = mpWindow->GetHeight()/2;
474                     SpaceObject * lookingAt = *mLookAtIterator;
475                     mpSpace->AddObjectToSpace(
476                                     new Planet("Vesta", //name
477                                                2.67e20, //mass
478                                                0.00625 , //radius
479                                                Coordinate( //position
480                                                   FromScale(
481                                                      (gPosX-hWidth)/hWidth) +
482                                                        lookingAt->
483                                                           GetPosition().GetX(),
484                                                   FromScale(
485                                                      (-gPosY+hHeight)/hHeight) +
486                                                        lookingAt->
487                                                           GetPosition().GetY()),
488                                                Coordinate( //velocity
489                                                    CalculateNewObjectSpeed(
490                                                       gPosX-hWidth, x-hWidth),
491                                                    CalculateNewObjectSpeed(
492                                                      -gPosY+hWidth, -y+hWidth)),
493                                                1.0, //colour red
494                                                0,  //colour green
495                                                0)); //colour blue
496                 }
497                 break;
498             }
499     }
500
501     /*****************************************************************************
502     * Getters and Setters
503     *
504     *****************************************************************************/
505     //  define the instance
506     Draw * Draw::mspInstance = 0;
507     void Draw::SetInstance(Draw * pInstance){
508         mspInstance = pInstance;
509     }
510
```

```
/*****************************************************************************
*   FILE: Space.h
*
*   FUNCTION: This class manages a space and all objects within it. It is the
*   class that calculates gravity between all objects and is also the class
*   that updates their positions depending on their speed and direction.
*
*   PURPOSE: The space class makes handling spaces and the objects within
*   it much easier and will also make it easier to use multiple spaces if
*   this is supported in the future.
*
*****************************************************************************/

#ifndef _Space_
#define _Space_

#include "SpaceObject.h"
#include "Star.h"
#include "Planet.h"
#include "Moon.h"
#include <list>

class Space{
    public:
    /** Constructors    **/
    //  constructs a space with the given integer as the amount of seconds to
    //  update when updating
    Space(int);
    /** Member Functions   **/
    //  adds a planet to the objects and planets lists
    void                        AddObjectToSpace(Planet*);
    //  adds a star to the objects and stars lists
    void                        AddObjectToSpace(Star*);
    //  adds a moon to the objects and moons lists
    void                        AddObjectToSpace(Moon*);
    //  removes the last element created
    void                        PopObjectFromSpace();
    //  calculates gravity between all elements in the objects list
    void                        CalculateGravity();
    //  calculates new positions for all elements in the objects list after a
    //  certain amount of time has passed
    void                        PassTime();
    /** Getters and Setters **/
    std::list<SpaceObject *>    GetObjectsInSpace()
                                    {return mObjectsInSpace;}
    std::list<Star *>           GetStarsInSpace()
                                    {return mStarsInSpace;}
    std::list<Planet *>         GetPlanetsInSpace()
                                    {return mPlanetsInSpace;}
    std::list<Moon *>           GetMoonsInSpace()
                                    {return mMoonsInSpace;}
    int                         GetTime()
                                    {return mTime;};
    void                        SetTime(int time)
                                    {mTime = time;}

    private:
    /** Class Members   **/
    std::list<SpaceObject *>    mObjectsInSpace;
    std::list<Planet *>         mPlanetsInSpace;
    std::list<Star *>           mStarsInSpace;
    std::list<Moon *>           mMoonsInSpace;
    int                         mTime;

};

#endif
```

```cpp
/*****************************************************************************
*   FILE: Space.cpp
*
*   FUNCTION: This class manages a space and all objects within it. It is the
*   class that calculates gravity between all objects and is also the class
*   that updates their positions depending on their speed and direction.
*
*   PURPOSE: The space class makes handling spaces and the objects within
*   it much easier and will also make it easier to use multiple spaces if
*   this is supported in the future.
*
*****************************************************************************/

#include "Space.h"

/*****************************************************************************
* Constructors
*
*****************************************************************************/

/**
    Name: Space(int)
    Function: Constructs a Space with the integer time given as time to pass
    in PassTime().
**/
Space::Space(int time){
    mTime = time;
}

/*****************************************************************************
* Member Functions
*
*****************************************************************************/

/**
    Name: AddObjectToSpace(Planet*)
    Function: Adds an object to the ObjectsInSpace list and the
    PlanetsInSpace list.
**/
void Space::AddObjectToSpace(Planet* pPlanet){
    mObjectsInSpace.push_back(pPlanet);
    mPlanetsInSpace.push_back(pPlanet);
}

/**
    Name: AddObjectToSpace(Moon*)
    Function: Adds an object to the ObjectsInSpace list and the
    MoonsInSpace list.
**/
void Space::AddObjectToSpace(Moon* pMoon){
    mObjectsInSpace.push_back(pMoon);
    mMoonsInSpace.push_back(pMoon);
}

/**
    Name: AddObjectToSpace(Star*)
    Function: Adds an object to the ObjectsInSpace list and the
    StarsInSpace list.
**/
void Space::AddObjectToSpace(Star* pStar){
    mObjectsInSpace.push_back(pStar);
    //  if there are less than 9 stars
    //  GLUT can only handle up to 8 light sources
    if(mStarsInSpace.size() < 9){
        pStar->SetLightSource(mStarsInSpace.size());
    }
    mStarsInSpace.push_back(pStar);

}

/**
    Name: PopObjectFromSpace()
    Function: Removes the last object from the ObjectsInSpace and the
    Star/Planet/Moon list depending on its type and frees the objects
    allocated memory.
**/
void Space::PopObjectFromSpace(){
    //  if there are objects in space
    if(!mObjectsInSpace.empty())
    {
        //  set a temp pointer to the last object in space
        SpaceObject* pLastObject = mObjectsInSpace.back();
        //  if there are stars in space
        if(!mStarsInSpace.empty())
        {
            //  set a temporary pointer to the last star in space
            Star* star = mStarsInSpace.back();
            //  if mass of star = mass of spaceobject
            if(pLastObject->GetMass() == star->GetMass() && star != 0)
            {
                //  remove the star from the list of stars
```

```
                mStarsInSpace.pop_back();
            }
        }
        //  if there are planets in space
        if(!mPlanetsInSpace.empty())
        {
            //  set a temporary pointer to the last planet in space
            Planet* planet = mPlanetsInSpace.back();
            //  if mass of planet = mass of spaceobject
            if(pLastObject->GetMass() == planet->GetMass())
            {
                //  remove the planet from the list of planets
                mPlanetsInSpace.pop_back();
            }
        }
        //  if there are moons in space
        if(!mMoonsInSpace.empty())
        {
            //  set a temporary pointer to the last moon in space
            Moon* moon = mMoonsInSpace.back();
            //  if mass of moon = mass of spaceobject
            if(pLastObject->GetMass() == moon->GetMass())
            {
                //  remove the moon from the list of moons
                mMoonsInSpace.pop_back();
            }
        }
        //  remove the last object from the list of objects
        mObjectsInSpace.pop_back();
        //  free the memory allocated by the object
        delete pLastObject;
    //  if the list is empty
    }else
    {
        //handle error
    }
}

/**
    Name: CalculateGravity()
    Function: Calculates gravity between all elements in ObjectsInSpace.
**/
void Space::CalculateGravity(){
    std::list<SpaceObject*>::iterator it;
    std::list<SpaceObject*>::iterator it2;
    SpaceObject * Object1;
    SpaceObject * Object2;
    //  iterate through the list
    for( it = mObjectsInSpace.begin(); it != mObjectsInSpace.end(); it++)
    {
        it2 = it;
        //  iterate one step ahead of the previous for
        for(it2++; it2 != mObjectsInSpace.end(); it2++)
        {
            Object1 = *it;
            Object2 = *it2;
            //  the universal gravitational constant
            const double g = 6.67428e-11;
            //  calculate the distance between the two objects
            Coordinate distance =
                Object2->GetPosition() - Object1->GetPosition();
            //  calculate the lentgh of the distance
            double length = distance.CalculateLength();
            //  calculate the gravitational pull between the objects:
            //  F = (G*m1*m2)/r*r where r is the length of the distance.
            double force =
                ((g*Object1->GetMass()*Object2->GetMass())/(length*length));
            distance = distance*(1/length);
            distance = distance*force;
            //  set the gravitational force to object1
            Object1->SetForce(Object1->GetForce()+distance);
            //  set the gravitational force to object2
            Object2->SetForce(Object2->GetForce()-distance);
        }
    }

}

/**
    Name: PassTime()
    Function: Calculates the new positions for all objects in space after a
    certain amount of time has passed.
**/
void Space::PassTime(){
    std::list<SpaceObject*>::iterator it;
    //  iterate through the list of objects in space
    for( it = mObjectsInSpace.begin(); it != mObjectsInSpace.end(); it++)
    {
        //  point to the current object
        SpaceObject * pObject = *it;
        //  if mass isnt zero. Reason: There is a division by mass later on.
```

```
183             if(pObject->GetMass() != 0)
184             {
185                 //  update position with current velocity
186                 pObject->SetPosition(pObject->GetPosition() +
187                                      (pObject->GetVelocity() * mTime));
188                 //  calculate acceleration using the current force
189                 Coordinate acceleration = Coordinate(pObject->GetForce() *
190                                                      (1/pObject->GetMass()));
191                 //  set new position using the acceleration
192                 pObject->SetPosition(pObject->GetPosition() +
193                                      ((acceleration)*(mTime*mTime)*0.5));
194                 //  set the new velocity using the acceleration
195                 pObject->SetVelocity(pObject->GetVelocity() +
196                                      acceleration*mTime);
197                 //  the calculation is finished so force is set to 0
198                 pObject->SetForce(Coordinate(0, 0));
199             }
200             else
201             {
202                 //handle error
203             }
204 
205         }
206     }
207 
```

```
/*****************************************************************************
*   FILE: Coordinate.h
*
*   FUNCTION: This class is made to handle coordinates in the two-dimensional
*   plane and can also perform some simple computations.
*
*   PURPOSE: To make it easier to handle coordinates within the code.
*
*****************************************************************************/

#ifndef _Coordinate_
#define _Coordinate_

class Coordinate{
    public:
    /** Constructors **/
    //  default constructor
    Coordinate(){};
    //  construct a coordinate out of two x and y doubles
    Coordinate(double, double);
    /** Member Functions    **/
    //  calculates the length of the coordinate
    double          CalculateLength();
    /** Getters and Setters **/
    double          GetX(){return mX;}
    double          GetY(){return mY;}
    void            SetX(double x){mX = x;}
    void            SetY(double y){mX = y;}
    /** Operator Overloads **/
    //  adds two coordinates
    Coordinate      operator+(Coordinate);
    //  subtracts from a coordinate with another coordinate.
    Coordinate      operator-(Coordinate);
    //  multiplies a coordinate with a double
    Coordinate      operator*(double);
    //  checks if two coordinates are unequal
    bool            operator!=(Coordinate);

    private:
    /** Class members **/
    double  mX;
    double  mY;
};

#endif
```

```cpp
/*******************************************************************************
*   FILE: Coordinate.cpp
*
*   FUNCTION: This class is made to handle coordinates in the two-dimensional
*   plane and can also perform some simple computations.
*
*   PURPOSE: To make it easier to handle coordinates within the code.
*
*******************************************************************************/

#include "Coordinate.h"
#include <math.h>

/*******************************************************************************
 * Constructors
 *
 *******************************************************************************/

/**
    Name: Coordinate(double, double)
    Function: Constructs a coordinate out of two doubles.
**/
Coordinate::Coordinate(double x, double y)
{
    mX = x;
    mY = y;
}

/*******************************************************************************
* Member Functions
*
*******************************************************************************/

/**
    Name: CalculateLength()
    Function: Caluclates the length of a coordinate as a straight line from
    origin.
**/
double Coordinate::CalculateLength()
{
    return sqrt(mX*mX + mY*mY);
}

/*******************************************************************************
* Operator Overloads
*
*******************************************************************************/

/**
    Name: operator+(Coordinate)
    Function: Adds two coordinates.
**/
Coordinate Coordinate::operator+(Coordinate toAdd)
{
    return Coordinate(mX+toAdd.GetX(),mY+toAdd.GetY());
}

/**
    Name: operator-(Coordinate)
    Function: Subtracts from a coordinate with another coordinate.
**/
Coordinate Coordinate::operator-(Coordinate toSubtract)
{
    return Coordinate(mX-toSubtract.GetX(),mY-toSubtract.GetY());
}

/**
    Name: operator*(double)
    Function: Multiplies a coordinate with a double.
**/
Coordinate Coordinate::operator*(double toMultiply)
{
    return Coordinate(mX*toMultiply,mY*toMultiply);
}

/**
    Name: operator!=(Coordinate)
    Function: Checks if two coordinates are unequal.
**/
bool Coordinate::operator!=(Coordinate toCompare)
{
    if(mX == toCompare.GetX() && mY == toCompare.GetY())
        return false;
    else
        return true;
}

```

```
/*****************************************************************************
*   FILE: SpaceObject.h
*
*   FUNCTION: This class is the base for all objects in space. All objects in
*   space have and need these properties. All objects in space are directly
*   or indirectly constructed using this class.
*
*   PURPOSE: By creating a base class for all objects in space it will be
*   easier to manage the objects in space since at least the members of this
*   class can always be accessed. This also means that it will be easier to
*   create different types of objects by just inheriting this class
*   and then adding the additional fucntionality in the new class.
*
*****************************************************************************/

#ifndef _SpaceObject_
#define _SpaceObject_

#include "Coordinate.h"
#include <string>

class SpaceObject{
    public:
    /** Constructors    **/
    //  default constructor
    SpaceObject(){};
    //  creates a SpaceObject with a name string, mass and radius
    //  doubles, position and velocit coordinates and RGB floats.
    SpaceObject(std::string, double, double, Coordinate, Coordinate, float,
                float, float);
    /** Getters and Setters **/
    std::string         GetName()
                            {return mName;}
    void                SetName(std::string name)
                            {mName = name;}
    double              GetRadius()
                            {return mRadius;}
    void                SetRadius(double radius)
                            {mRadius = radius;}
    double              GetMass()
                            {return mMass;}
    void                SetMass(double mass)
                            {mMass = mass;}
    Coordinate          GetPosition()
                            {return mPosition;}
    void                SetPosition(Coordinate position)
                            {mPosition = position;}
    Coordinate          GetVelocity()
                            {return mVelocity;}
    void                SetVelocity(Coordinate velocity)
                            {mVelocity = velocity;}
    Coordinate          GetForce()
                            {return mForce;}
    void                SetForce(Coordinate force)
                            {mForce = force;}
    float               GetRed()
                            {return mRed;}
    float               GetGreen()
                            {return mGreen;}
    float               GetBlue()
                            {return mBlue;}
    void                SetColour(float red, float green, float blue)
                            {mRed = red; mGreen = green; mBlue = blue;}

    protected:
    /** Class Members   **/
    std::string         mName;
    double              mRadius;
    double              mMass;
    Coordinate          mPosition;
    Coordinate          mVelocity;
    Coordinate          mForce;
    float               mRed;
    float               mGreen;
    float               mBlue;
};

#endif
```

```cpp
/*****************************************************************************
*   FILE: SpaceObject.cpp
*
*   FUNCTION: This class is the base for all objects in space. All objects in
*   space have and need these properties. All objects in space are directly
*   or indirectly constructed using this class.
*
*   PURPOSE: By creating a base class for all objects in space it will be
*   easier to manage the objects in space since at least the members of this
*   class can always be accessed. This also means that it will be easier to
*   create different types of objects by just inheriting this class
*   and then adding the additional fucntionality in the new class.
*
*****************************************************************************/

#include "SpaceObject.h"

/*****************************************************************************
 * Constructors
 *
 *****************************************************************************/
/**
    Name: SpaceObject(std::string, double, double,
                      Coordinate, Coordinate,
                      float, float, float)
    Function: Creates a SpaceObject with a name string, mass and radius
    doubles, position and velocit coordinates and RGB floats.
**/
SpaceObject::SpaceObject(std::string name, double mass, double radius,
                         Coordinate position, Coordinate velocity,
                         float red, float green, float blue)
{
    mName = name;
    mMass = mass;
    mRadius = radius;
    mPosition = position;
    mVelocity = velocity;
    //  force starts at neutral
    mForce = Coordinate(0, 0);
    mRed = red;
    mGreen = green;
    mBlue = blue;
}

```

```
/*****************************************************************************
*   FILE: Star.h
*
*   FUNCTION: This class inherits SpaceObject and is used to create
*   stars in space. Stars are different from other space objects because they
*   emit light.
*
*   PURPOSE: The star class exists to create space objects that emit light.
*
*****************************************************************************/

#ifndef _Star_
#define _Star_

#include "SpaceObject.h"

class Star : public SpaceObject{
    public:
    /** Constructors    **/
    //  constructs a star with a name string, mass double, radius double,
    //  position coordinate, velocity coordinate and RGB floats
    Star(std::string, double, double,
         Coordinate, Coordinate,
         float, float, float);
    /** Getters and Setters **/
    unsigned int GetLightSource()
                    {return mLightSource;}
    void SetLightSource(unsigned int sourceId)
                    //  16384 is the integer where the glut enumerators for light
                    //  sources start
                    {mLightSource = 16384+sourceId;}
    private:
    /** Class Members   **/
    unsigned int mLightSource;
};

#endif
```

```cpp
/*****************************************************************************
*   FILE: Star.cpp
*
*   FUNCTION: This class inherits SpaceObject and is used to create
*   stars in space. Stars are different from other space objects because they
*   emit light.
*
*   PURPOSE: The star class exists to create space objects that emit light.
*
*****************************************************************************/

#include "Star.h"

/*****************************************************************************
 * Constructors
 *
 *****************************************************************************/

/**
    Name: Star(std::string, double, double, Coordinate, Coordinate,
                  float, float, float)
    Function: Creates a star with a name string, mass and radius doubles,
    position and velocity coordinates and RGB floats.
**/
Star::Star(
         std::string name, double mass, double radius,
         Coordinate position, Coordinate velocity,
         float red, float green, float blue)
{
    mName           = name;
    mMass           = mass;
    mRadius         = radius;
    mPosition       = position;
    mVelocity       = velocity;
    mForce          = Coordinate(0, 0); //force starts at neutral
    mRed            = red;
    mGreen          = green;
    mBlue           = blue;
}

```

```
/*****************************************************************************
*   FILE: Planet.h
*
*   FUNCTION: This class inherits SpaceObject and is used to create
*   planets in space.
*
*   PURPOSE: Instead of using the SpaceObject class for constructing planets,
*   having a class dedicated for constructing planets will make it easier to
*   give planets special properties later on. This was the code will also be
*   easier to understand, since a constructed planet is a planet and a
*   SpaceObject can be any object in space, including suns and moons.
*
*****************************************************************************/

#ifndef _Planet_
#define _Planet_
#include "SpaceObject.h"

class Planet : public SpaceObject{
    public:
    /** Constructors    **/
    //  constructs a planet with a name string, mass and radius doubles,
    //  position and velociy coordinates and RGB floats
    Planet(std::string, double, double,
           Coordinate, Coordinate,
           float, float, float);
};
#endif

```

```cpp
/*****************************************************************************
*   FILE: Planet.cpp
*
*   FUNCTION: This class inherits SpaceObject and is used to create
*   planets in space.
*
*   PURPOSE: Instead of using the SpaceObject class for constructing planets,
*   having a class dedicated for constructing planets will make it easier to
*   give planets special properties later on. This was the code will also be
*   easier to understand, since a constructed planet is a planet and a
*   SpaceObject can be any object in space, including suns and moons.
*
*****************************************************************************/

#include "Planet.h"

/*****************************************************************************
 * Constructors
 *
 *****************************************************************************/

/**
    Name: Planet(std::string, double, double, Coordinate, Coordinate,
                 float, float, float)
    Function: Creates a planet with a name string, mass and radius doubles,
    position and velocity coordinates and RGB floats.
**/
Planet::Planet(
               std::string name, double mass, double radius,
               Coordinate position, Coordinate velocity,
               float red, float green, float blue){

    mName           = name;
    mMass           = mass;
    mRadius         = radius;
    mPosition       = position;
    mVelocity       = velocity;
    //  force starts at neutral
    mForce          = Coordinate(0, 0);
    mRed            = red;
    mGreen          = green;
    mBlue           = blue;
}

```

```
/*****************************************************************************
*   FILE: Moon.h
*
*   FUNCTION: This class inherits SpaceObject and uses the distance from
*   its owner instead of position and velocity coordinates. It then
*   calculates its own velocity and position with the help of the distacne to
*   its owner.
*
*   PURPOSE: To make it easier to create moons. If owner/satellite pointers
*   are included later on it might add extra functionality to the
*   application.
*
*****************************************************************************/

#ifndef _Moon_
#define _Moon_

#include "Planet.h"
#include "SpaceObject.h"

class Moon : public SpaceObject{
    public:
    /** Constructors    **/
    //  constructs a moon with a name string, mass, radius and distance
    //  doubles as well as RGB floats
    Moon(Planet*, std::string, double, double,
         double,
         float, float, float);
};
#endif

```

```cpp
/*************************************************************************
*   FILE: Moon.cpp
*
*   FUNCTION: This class inherits SpaceObject and uses the distance from
*   its owner instead of position and velocity coordinates. It then
*   calculates its own velocity and position with the help of the distacne to
*   its owner.
*
*   PURPOSE: To make it easier to create moons. If owner/satellite pointers
*   are included later on it might add extra functionality to the
*   application.
*
*************************************************************************/

#include "Moon.h"
#include <math.h>

/*************************************************************************
 * Constructors
 *
 *************************************************************************/


/**
    Name: Moon(Planet*, std::string, double, double, double,
                float, float, float)
    Function: Creates a moon with a position and speed depending on the
    distance from its owner. Moons will always orbit their owner.
**/
Moon::Moon(Planet * pOwner, std::string name, double mass, double radius,
           double distance, float red, float green, float blue)
{
    mName = name;
    mMass = mass;
    mRadius = radius;
    //  place the moon at a certain distance from its owner
    mPosition = Coordinate(pOwner->GetPosition().GetX(),
                           pOwner->GetPosition().GetY()+distance);
    /*  START Calculate velocity    */
    //  the gravitational constant
    const double g = 6.67428e-11;
    const double PI = 3.1415926;
    //  calculating orbital period: T = 2*PI*sqrt(a*a*a/(g*M))
    //  where M is the mass of the central object and a is the distance
    double a = distance*distance*distance;
    double T = 2*PI*sqrt(a/(g*pOwner->GetMass()));
    //  calculating orbital speed: v = (a*2*PI)/T
    //  where a is the distance and T is the orbital period
    double circumference = distance*2*PI;
    double velocity = circumference/T;
    //  setting the speed
    mVelocity = Coordinate(pOwner->GetVelocity().GetX()+velocity,
                           pOwner->GetVelocity().GetY());
    /*  END Calculate velocity  */
    //  force starts at neutral
    mForce = Coordinate(0, 0);
    mRed = red;
    mGreen = green;
    mBlue = blue;
}

```