

|  |
| --- |
| Donic the Ledgehog |
|  |
| ICT289 Assignment 2 Report  Christo Stephenson – 33437999 Kye Horbury – 31900516 Rory Lowe-McLoughlin – 33274819 |

Table of Contents

[Project Description 3](#_Toc10989330)

[Program Structure 4](#_Toc10989331)

[General client program structure 4](#_Toc10989332)

[Game objects 4](#_Toc10989333)

[Use of game objects in main program 5](#_Toc10989334)

[Physics Simulations 8](#_Toc10989335)

[General ball movement 8](#_Toc10989336)

[Ball movement from player interaction 8](#_Toc10989337)

[External forces 8](#_Toc10989338)

[Collision detection 9](#_Toc10989339)

[Platform and wall collision detection 9](#_Toc10989340)

[Coin collision detection 12](#_Toc10989341)

[Animation 13](#_Toc10989342)

[Ball animation 13](#_Toc10989343)

[Camera 13](#_Toc10989344)

[Moving platforms 14](#_Toc10989345)

[Moving walls 15](#_Toc10989346)

[Disappearing platforms 15](#_Toc10989347)

[Coins 17](#_Toc10989348)

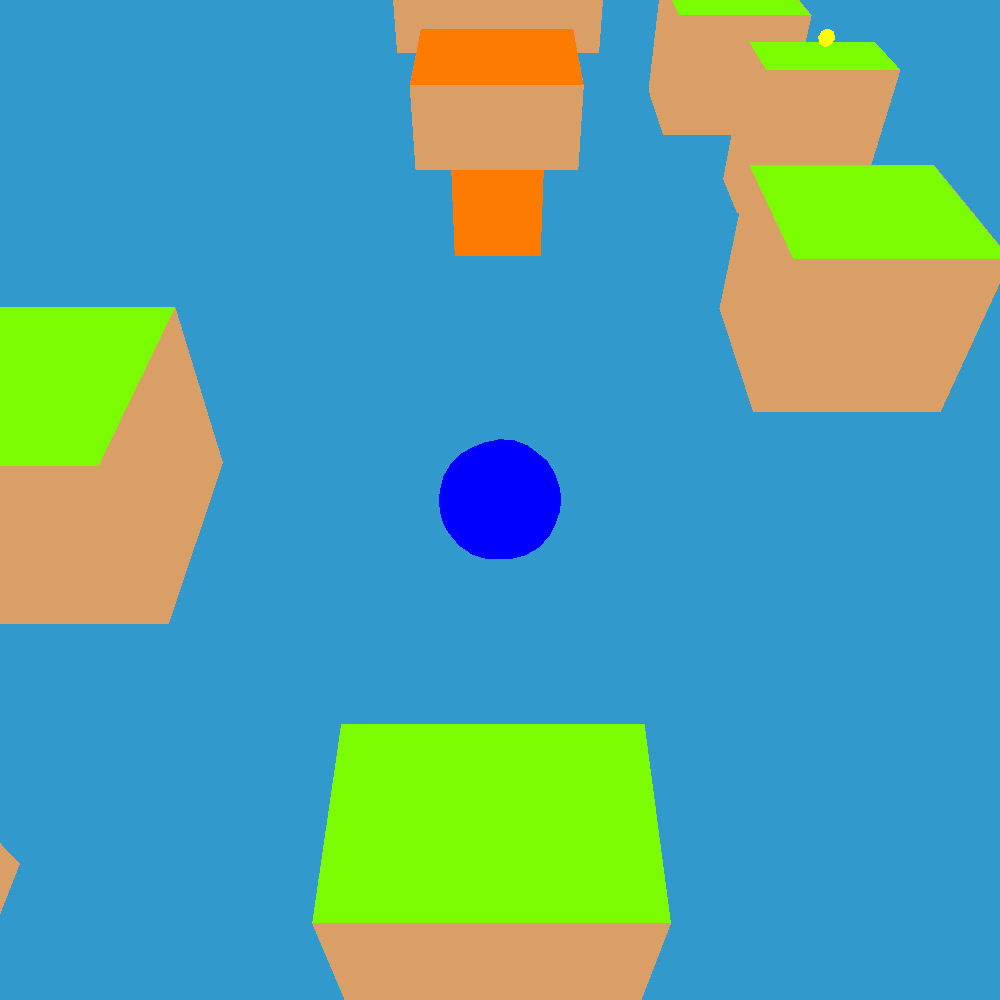
[User Guide 18](#_Toc10989349)

[Program setup 18](#_Toc10989350)

[User controls 18](#_Toc10989351)

# Project Description

Donic the Ledgehog is a game created in C++ using the OpenGL libraries that aims to provide a simulated 3D physical environment where the user controls and navigates a ball through a modelled obstacle course. The game simulates several physical aspects, such as gravity, wind resistance, and object collisions. It also includes a coin collection and scoring system. The game lends itself to classic platformer-style games where the element of challenge is to dexterously jump platform to platform. As the game progresses, various different platform-related challenges are presented to the player, each animated and calculated using realistic physics calculation.



Screenshot of Donic the Hedgehog

# Program Structure

## General client program structure

The client program primarily consisted of a main() function to control the flow of the program and an init() function to initialize the program. The animate() function, which is called with the glutTimerFunc() function repeatedly, performs animations based on user inputs, external forces applied to the ball and collision detections. A display() function to handle drawing the scene of the program and a resetCourse() method used to reset values which have been changed during the execution of the program. Naturally, other helper methods were also in the main program.

A header file for the client program called mainHeader.cpp contained many global variables, objects and constants used throughout the client program. Using these in the global scope was necessary due to the required use of function pointers due to the nature of OpenGL programs.

The structure and flow of the program focuses on a single ball, which is stored as a struct in mainHeader.h as a global struct called “ball”.

## Game objects

Any content that is not the ball in the program is defined as a game object. In the code, a game object is any object that extends the abstract class Object. This abstract class requires a game object to have an animate method that tells the game object the ball’s position, radius and the milliseconds passed since the game’s last tick. It also requires a display method, in which the game object draws all its own polygons, lines, etc. that it wants to be displayed. The getBallVel() method determines what velocity the game object thinks the ball should have, especially in the case of a collision. The collisionDetected method simply returns a Boolean of whether the ball in the program has collided with the game object. The touchingFloor() method returns a Boolean of whether or not the ball is utilizing the game object as a floor (e.g. the ball is sitting on top of the game object). The reset() method is used to reset an object to its initial state. The game object abstract class specification is listed below.

|  |
| --- |
| class Object |
| { |
| public: |
| virtual void animate(const point3D& currPos, float radius, float t) = 0; |
|  |
| virtual void display() = 0; |
|  |
| virtual point3D getBallVel(const point3D& vel, const point3D& prevPos, float radius) = 0; |
|  |
| virtual bool collisionDetected(const point3D& vel, const point3D& prevPos, float radius) = 0; |
|  |
| virtual bool touchingFloor(const point3D& currPos, float radius) = 0; |
|  |
| virtual void reset() = 0; |
| }; |

Typically, the realization of game objects have default constructors deleted and include parameterized constructors to determine the values of the game object (e.g. 3D points of the object, speed of the object, etc.).

## Use of game objects in main program

Game objects are constructed and stored in the mainHeader.cpp globally. A vector of object pointers are stored in the same place as the game objects. The pointers to the game objects are added to the vector in the init() function in the client program.

The display() methods of the game objects are all called in the display() function in main.cpp (code snippet below).

|  |
| --- |
| for (unsigned i = 0; i < objects.size(); i++) |
| objects[i]->display(); |

The reset() methods of game objects are also called in a similar fashion to above in the resetCourse() function in main.cpp.

In the animate() function in main.cpp, the animate() function for each game object is called (code snippet below).

|  |
| --- |
| for (unsigned i = 0; i < objects.size(); i++) |
| objects[i]->animate(ball.currPos, ball.radius, deltaT\_seconds); |

Later in the same animate() function in main.cpp, after the ball’s velocity has been calculated based on player input and external forces, the methods from the game objects are used to calculate collision detections. The vector holding the pointers to game objects are iterated through. Each iteration checks if a game object has a collision with the collisionDetected() method. If the method returns true, the getBallVel() method of the game object is called and the return value is noted. If any collision is detected, the ball’s velocity is set to the average ball velocity calculated by the various game objects the ball collided with (code snippet below).

|  |
| --- |
| for (unsigned i = 0; i < objects.size(); i++) |
| { |
| if (objects[i]->collisionDetected(ball.currVel, ball.prevPos, ball.radius)) |
| { |
| tempVel = objects[i]->getBallVel(ball.currVel, ball.prevPos, ball.radius); |
|  |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
| collisions++; |
| } |
| } |
|  |
| //Calc average velocity of collisions and change velocity of ball if collisions happened |
| if (collisions > 0) |
| { |
| //Determine average |
| colVel.x /= (float) collisions; |
| colVel.y /= (float) collisions; |
| colVel.z /= (float) collisions; |
|  |
| //Modify velocity |
| ball.currVel.x = colVel.x; |
| ball.currVel.y = colVel.y; |
| ball.currVel.z = colVel.z; |
| } |

The touchingFloor() method of game objects is also utilized. This is utilized in the keyboard() function in main.cpp to make sure to only initialize a jump when touching a game object as a floor (in addition to checking that a jump has not already been initialized). Code snippet below.

//Detects if spacebar is pressed

|  |
| --- |
| if (key == 32) |
| { |
| ball.moveDir.posY = false; |
| //Detects if jump can be commenced |
| for (unsigned i = 0; i < objects.size() && ball.moveDir.posY == false; i++) |
| { |
| ball.moveDir.posY = objects[i]->touchingFloor(ball.currPos, ball.radius); |
| } |
|  |
| //Set start height if starting to jump |
| if (ball.moveDir.posY) |
| { |
| ball.jumpStartH = ball.currPos.y; |
| } |
| } |

# Physics Simulations

## General ball movement

The ball’s coordinates is determined by taking the ball’s previous coordinates from the program’s last tick and adding on the ball’s current calculated velocity, which is determined from a variety of factors. To start off with, the current velocity is initially set to the ball’s previous velocity from the previous tick.

## Ball movement from player interaction

A factor is the player’s movement of the ball. In the case of moving the ball left, right, forwards and backwards, this is determined by movementAcceleration \* deltaT on both the X and Z axis. In the case of moving the ball up from jumping (assuming that the input is validated by the keyboard() function), the calculation is similar. The equation is jumpAcceleration \* deltaT on the Y axis, however this is only applied if the ball has not reached its maximum jump height. If the ball has reached its maximum jump height, the program sets player jump movement to false. Code snippet below.

|  |
| --- |
| if (ball.moveDir.posY == true && (ball.currPos.y < ball.jumpStartH + ball.jumpH)) |
| ball.currVel.y += ball.jumpAcc \* deltaT\_seconds; |
| else |
| ball.moveDir.posY = false; |

These ball movement factors are added/subtracted onto the ball’s velocity.

## External forces

A factor that is calculated into the ball’s movement is the external acceleration forces applied to the ball. This may include things such as gravity and wind. This is simply calculated on each axis by externalAcceleration \* deltaT, where deltaT is the milliseconds difference between the previous tick and the current. This value is added onto the current velocity.

Wind resistance is also calculated. This is done in the windResistance() function in main.cpp, which simply applies a coefficient to the ball’s current velocity that should be lower than 1 to slow down the ball’s velocity.

## Collision detection

### Platform and wall collision detection

While game objects do provide collision detection, inside game objects lies AxisAlignedPlanes. AxisAlignedPlane is an abstract class which has the children XAlignedPlane, YAlignedPlane, ZAlignedPlane for the respective axis. These are 2D planes that are aligned with an axis. A code snippet is provided below specifically for the YAlignedPlane detailing the collision detection algorithm. It should be noted that axis2 is the X axis and axis3 is the Z axis, while mainAxis is the Y axis. What axis is which changes in XAlignedPlane and ZAlignedPlane.

|  |
| --- |
| //Check if main axis aligns |
| if (newPos.y <= mainAxis + radius && newPos.y >= mainAxis - radius) |
| { |
| //Check if secondary axis aligns (generously) |
| if ((newPos.x - (radius / X\_Z\_COLLISION\_MARGIN)) <= std::max(axis2Min, axis2Max) && |
| (newPos.x + (radius / X\_Z\_COLLISION\_MARGIN)) >= std::min(axis2Min, axis2Max) ) |
| { |
| //Check if third axis aligns (generously) |
| if ((newPos.z - (radius / X\_Z\_COLLISION\_MARGIN)) <= std::max(axis3Min, axis3Max) && |
| (newPos.z + (radius / X\_Z\_COLLISION\_MARGIN)) >= std::min(axis3Min, axis3Max) ) |
| { |
| detected = true; |
| } |
| } |
| } |

As seen above, the collision detection checks if the ball is within the main axis’ reach. Then if checks if the ball is within the second and third axis’ range with some margin of error to spare (of which the YAlignedPlane is the most generous because players would be upset if they fell through what they thought was a solid floor).

See the code snippet below.

|  |
| --- |
| point3D YAlignedPlane::getBallVel(const point3D& vel, const point3D& prevPos, float radius) |
| { |
| point3D newVel = vel; |
| //Inverts velocity and applies bounce coefficient, if collision detected |
| if (collisionDetected(vel, prevPos, radius)) |
| { |
| newVel.y = -(bounceCoefficient \* vel.y); |
| } |
|  |
| return newVel; |
| } |

The above snippet details how the YAlignedPlane determines a ball’s velocity. What it does is it detects if a collision is detected, and if it is detected it inverts the velocity within the axis in question and applies the bounce coefficient to it so the collision effectively absorbs the ball’s energy or gives the ball energy (in the project, the bounce coefficient is always set to 0 in the game object’s constructor as the ball bouncing on surfaces is not a desired trait).

Of course, since there are many AxisAlignedPlanes, many of the ball velocities they return will be conflicting. The solution to this problem is to determine the average velocity from collisions if a collision occurs. See the code snippet below to see how this is calculated from a basic platform made of 5 Axis Aligned Planes.

|  |
| --- |
| if (xPlane1.collisionDetected(vel, prevPos, radius)) |
| { |
| tempVel = xPlane1.getBallVel(vel, prevPos, radius); |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
|  |
| collisions++; |
| } |
|  |
| if (xPlane2.collisionDetected(vel, prevPos, radius)) |
| { |
| tempVel = xPlane2.getBallVel(vel, prevPos, radius); |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
|  |
| collisions++; |
| } |
|  |
| if (yPlane.collisionDetected(vel, prevPos, radius)) |
| { |
| tempVel = yPlane.getBallVel(vel, prevPos, radius); |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
|  |
| collisions++; |
| } |
|  |
| if (zPlane1.collisionDetected(vel, prevPos, radius)) |
| { |
| tempVel = zPlane1.getBallVel(vel, prevPos, radius); |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
|  |
| collisions++; |
| } |
|  |
| if (zPlane2.collisionDetected(vel, prevPos, radius)) |
| { |
| tempVel = zPlane2.getBallVel(vel, prevPos, radius); |
| colVel.x += tempVel.x; |
| colVel.y += tempVel.y; |
| colVel.z += tempVel.z; |
|  |
| collisions++; |
| } |
|  |
| //Return average collision velocity if collisions happened |
| if (collisions > 0) |
| { |
| //Determine average |
| colVel.x /= (float) collisions; |
| colVel.y /= (float) collisions; |
| colVel.z /= (float) collisions; |
|  |
| return colVel; |
| } |
| else |
| //Return input velocity if no collisions occured |
| { |
| colVel = vel; |
| return colVel; |
| } |

To see how the velocity is calculated after colliding with one or more objects, please refer to “Use of game objects in main program” section under the “Program Structure” chapter.

### Coin collision detection

Coins are only meant to appear if they have not been collided with. This collision detection is simply done with bounding spheres. It checks if the distance between the centre of the coin and the centre of the ball combined is equal to or smaller than the radius of the coin and the radius of the ball combined. If the check returns true, the collision is detected, and the coin is set to never appear again until the game is reset.

# Animation

## Ball animation

The animation of the ball is controlled by the animation function which is called at regular and frequent intervals. The time between each animation call is registered as *deltaT* and is the basis for many calculations. For the animation of the ball, the *deltaT* is first converted into milliseconds and then the ball is checked to see if it is moving in either or both, X and Z directions. For example, if moving in the X direction, then the ball object’s rotation axis is set to Y (if it were moving in the Z direction, the rotation axis would be set to X). The angle to be rotated by is then calculated by multiplying the current velocity in the direction of movement to the milliseconds passed and then adding that to the previously stored rotation angle. This means the ball will continue to rotate until movement in a direction ceases, and then the rotation angle is reset to 0. When the ball’s display function, *drawBall()* is called in the animation process, the ball is rotated using *glRotatef()* and the values previously calculated.

## Camera

Donic the Ledgehog’s camera system is closely tied to the ball’s movement. It is controlled by the *gluLookAt()* function which is called in the *display()* function, meaning the look at coordinates are updated on every animation tick of the program. The parameters given to *gluLookAt()* are the ball’s current x, y, and z position. The camera’s base is set to the ball’s position but with 1000 added to the y component and 1000 subtracted from the z component. This means the camera’s base is vertically up and horizontally behind the ball. The look at parameters are just the ball’s current x, y and z position, so the ball is always centred in the screen and the camera moves with the ball.

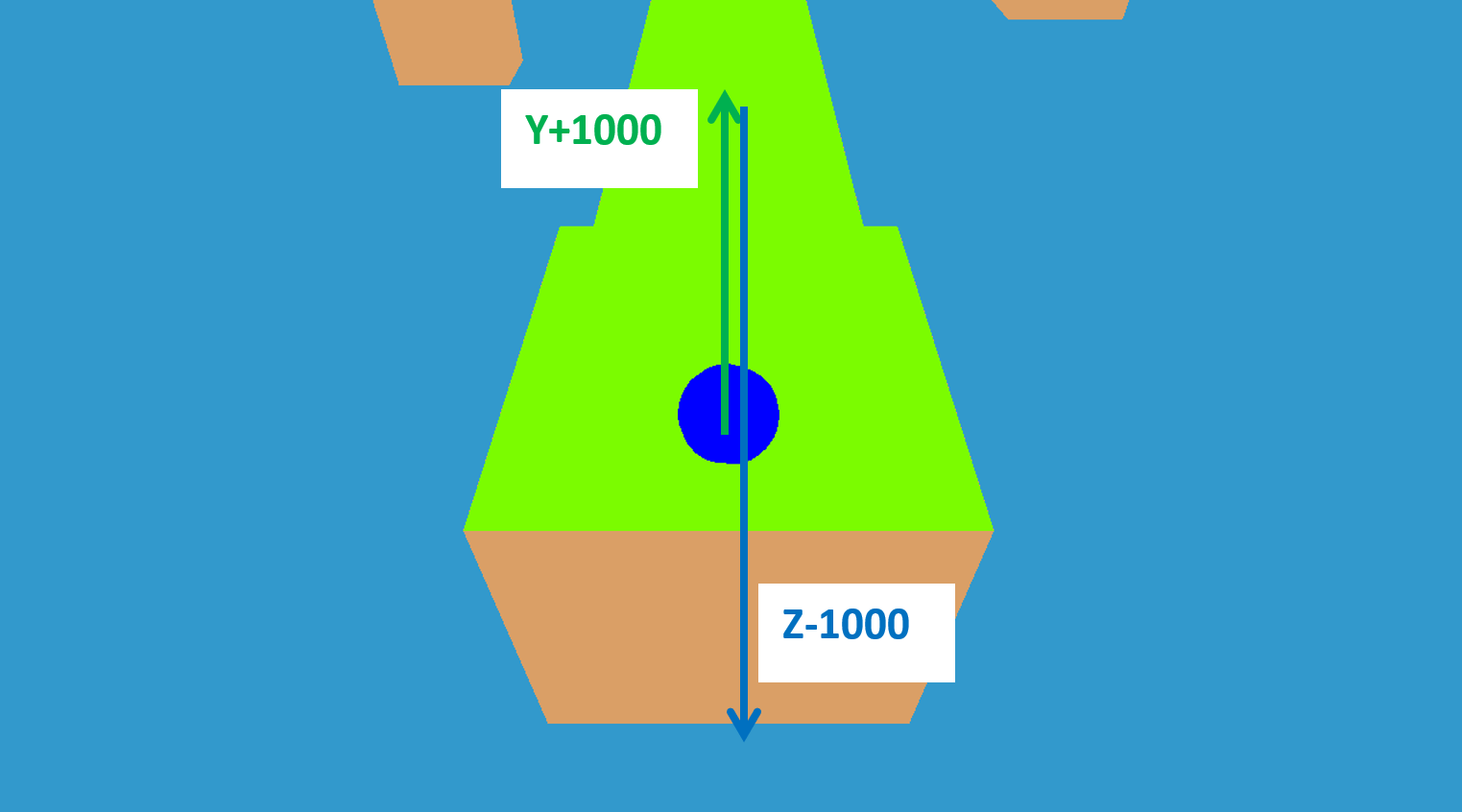


illustration of relative camera position

## Moving platforms

On creation, moving platforms are assigned a maximum distance they should travel from their origin in the x plane in which they move as well as a Boolean for which direction they are currently moving in, when the platform is constructed. Also given to the constructor is the platform’s speed per millisecond and original coordinates of its corners. In the main *animate()* function, each object’s respective *animate()* function is called and the *deltaT* is passed in. First, the function checks if the platform’s position is already beyond its maximum distance value. If so, then the directional Boolean is inverted. Then the new amount of movement relative to amount of time passed is calculated by multiplying together the platform’s speed value (defined in constructor) and the *deltaT*. This value is either added or subtracted to platform’s corner coordinates depending on the direction of movement. Every time *display()* is called, the platform will draw itself with new coordinates, animating the platform. These coordinates are also used for collision detection in another function for a continually updated collision system.

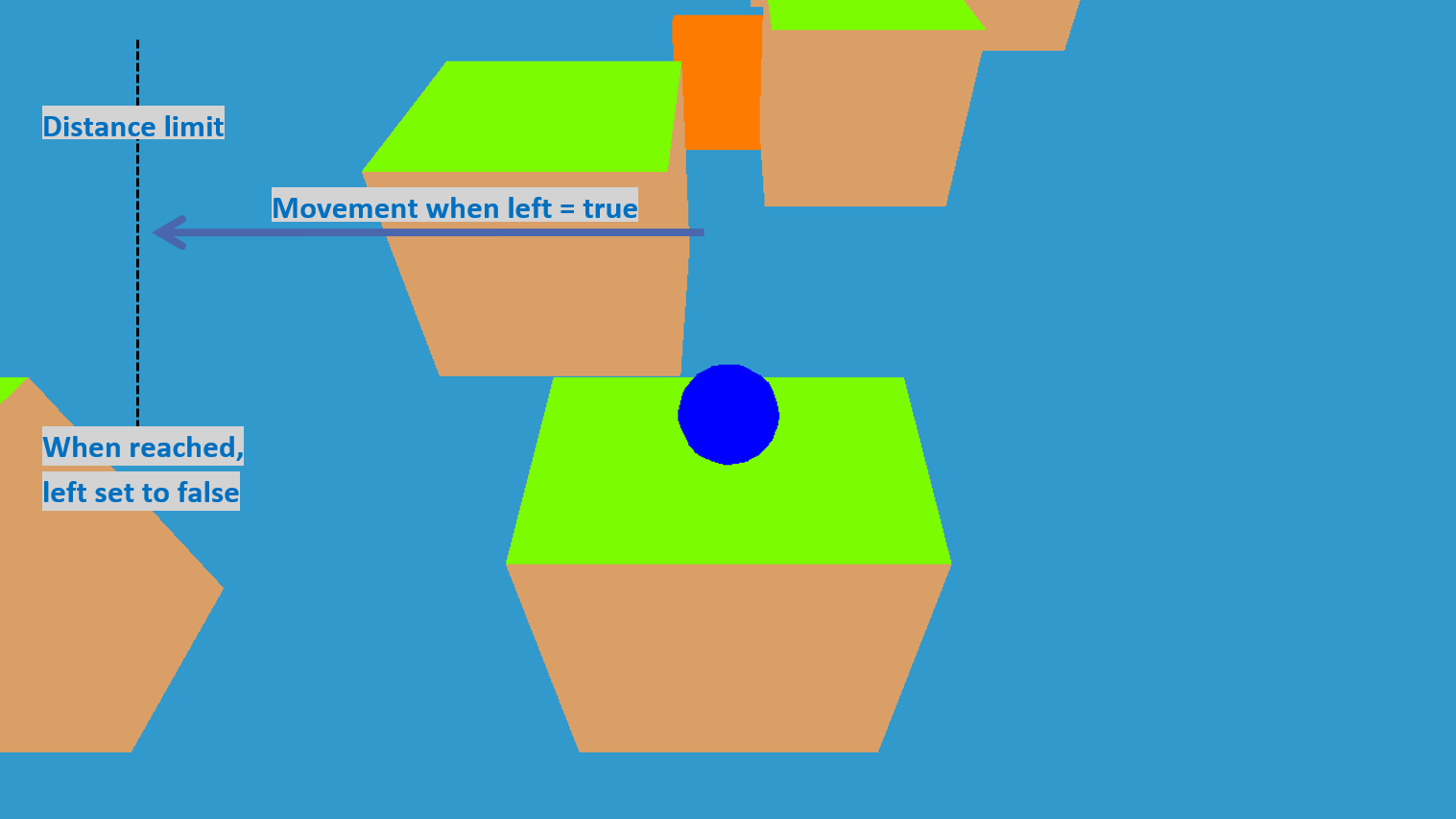


Illustration of platform movement

## Moving walls

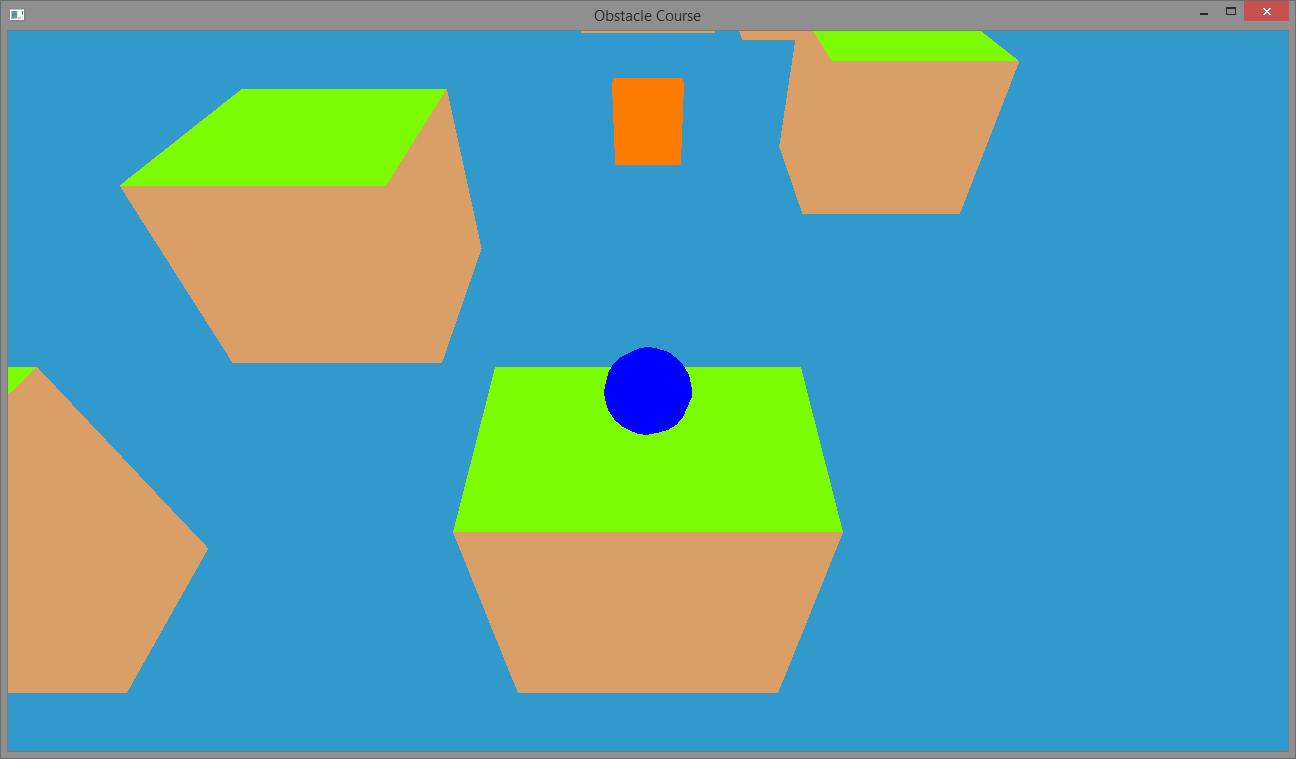
Moving walls are animated in the same way moving platforms are, but they move in the y axis rather than the x axis. They are constructed in the same way and have the same directional checks, movement calculation and axis aligned plane manipulation.

## Disappearing platforms

Disappearing platforms are assigned a *timeToLive* value that defines how long they remain before they disappear after being touched. The main *animate()* function passes in the ball’s current position, radius and the *deltaT* to the disappearing platform’s *animate()* function. From there the *timeToLive* value is checked to be above 0 and if successful then the *touchingFloor()* function is called, using the ball’s position and radius to determine if the ball has touched the disappearing platform. If it has been touched, a Boolean is used to determine the current colour of the flashing disappearing platform. First a timer is started, counting the amount of *deltaT* since the colour last changed. Once it reaches a defined threshold, the colour Boolean is inverted, the timer reset to 0 and begins counting again.

In the disappearing platform’s *display()* function, it is only drawn if it’s *timeToLive* is above 0 and the colour it is drawn in is dependent on the Boolean previously assigned.

The disappearing platform’s collision detection function is also only called when the *timeToLive* is above 0.

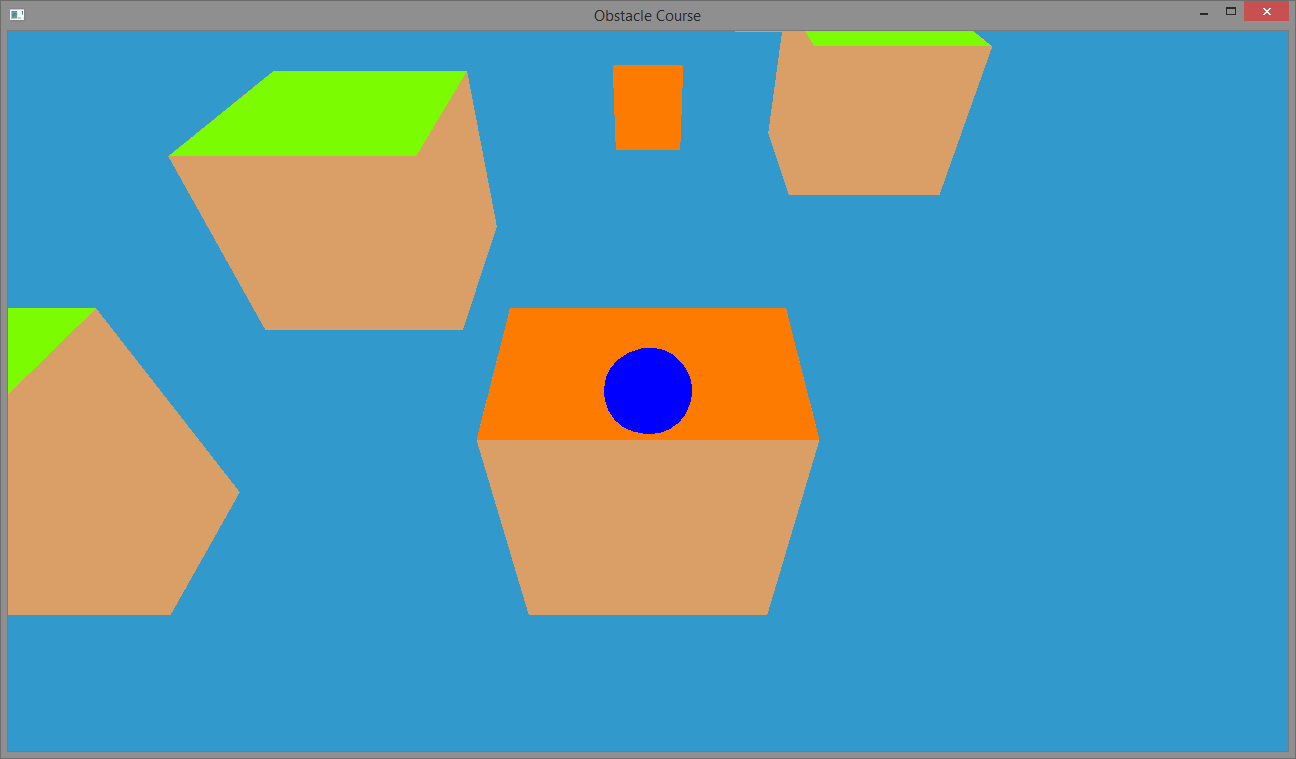


TouchingFloor() returns true

lastColourChange timer = 0

lastColourChange timer+= deltaT

timeToLive+=deltaT



lastColourChange timer = threshold

Boolean Colour = !Colour

lastColourChange timer =0

lastColourChange timer += deltaT

timeToLive+=deltaT

## Coins

A coin’s animation is called similarly to all other game objects. In the *animate()* function, each object is iterated through and has its relevant *animate()* function called, separate to the *animate()* function in main. A coin has a Boolean *collected* value that is initiated to false. In the coin’s *animate()* function, if *collected* is equal to false (if the coin has not been collected), then a collision check is run. This collision check takes the ball’s current position and radius which were passed in as parameters and checks the coin’s position, relative to the ball’s position and it’s radius for each axis. If a collision is found to have occurred, the *collected* Boolean is set to true which causes it’s display function to not be called, making the coin disappear on touch.

# User Guide

## Program setup

1. Open the project folder
2. Ensure it contains all .cpp and .h files as well as the three .raw files in the images folder
3. Open AssignmentTwo.cbp in Code::Blocks
4. Ensure all .cpp and .h files are successfully linked to the project
5. Ensure relevant glut & OpenGL libraries have been linked in global compiler settings
6. Click ‘Build and Run’

## User controls

|  |  |
| --- | --- |
| **Keyboard:** |  |
| W | Move Forward |
| A | Move Left |
| S | Move Back |
| D | Move Right |
| Space | Jump |
| M | Moon-Jump cheat mode |
| N | Toggle wind resistance |
| Q | Reset Level |
| P | Quit |