NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC
☎: 08258 - 281039 – 281263, Fax: 08258 – 281265
Department of Computer Science and Engineering
B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

Report on Mini Project

"Cricket Simulation"

Course Name: Computer Graphics and Multimedia

Course Code:16CS703

Semester: 7 Section: C

Submitted to,

Mr. Pradeep Kanchan
Asst Prof Gd III
Dept. of CSE, NMAMIT

Submitted by:

Rahul D Shetty	**4NM16CS111**
Rajath Gundmi Aithal	**4NM16CS112**
Rakesh Arasu	**4NM16CS113**

Date of submission:
04-11-2019

Signature of Course Instructor

Abstract

This project is focused on implementing a cricket simulation using computer graphics. We use OpenGL to recreate a scenario on a cricket pitch from the ball being bowled, the batsman striking the ball and the result of the delivery. We try to recreate all the real-world scenarios and circumstances and show all possible results.

Table of Contents

Introduction

The game of cricket is one spanning many decades and is a matter of pride and joy to our nation. It involves 11 players from each team (2 teams per match) battling it out on a circular/oval ground. The rules are as follows:

- One team gets to bowl first while the other bats. This is decided by means of a coin toss.
- The batting team has 2 batsmen on the field at a time while the fielding team has 11 players which include 1 bowler, 1 wicketkeeper and 9 fielders.
- The bowling team attempts to dismiss all batsmen while the batting team must chase down the target set by the bowling team.

In this project, we attempt to display this beautiful game and its components using OpenGL. In this demonstration, we will focus more on the batting side of events and the scenarios that occur from their viewpoint. The possible events that can occur in a single delivery are:

- Six - A ball is considered to be hit for a six if it crosses the boundary line of the field without bouncing anywhere on the ground *i.e*. it must directly leave the field of play after coming into contact with the bat of the batsman
- Four - A four or boundary is said to be scored if the ball crosses the boundary line after bouncing one or more times after coming into contact with the ball.
- 1's - This is scored when the batsman after hitting the ball manages to run from one end of the pitch to the other and the number of runs is equal to the number of times they traverse the entire length of the pitch.
- Out - When the bowler takes down a wicket or the ball struck by the batsman is caught by any fielder without hitting to the ground then that batsman has lost a wicket for his team.

Implementation Details

The program uses the OpenGL library to draw shapes on the screen using its different inbuilt functions. The entire code is written in C programming language and each of the logic is modularized into different units. The simulation of the cricket game consists of different scenes like taking a run, hitting 4 and 6 and so on. In order to maintain a smooth transition between all different states, we have used the concept of storing this information in a variable and drawing the appropriate scene in the *display()* method. The following code is used for switching between different scenes starting from 0 (Main Menu):

```
function display(){
        // This function is used to draw a shape on screen.
        if (app_state == 0) do:
                drawMainMenu();
        else if (app_state == 1) do:
                drawBattingScene();
        else if (app_state == 2) do:
                drawRunScene();
}
```

To switch between different scenes in the application we use keystrokes and depending on which keystroke has been pressed when the *app_state* is changed. The following code snippet shows how different keyboard strokes have been maintained for changing the state. The functions *handleMenuKey*, *handleBattingKey*, and

```
function keyboard(key, x, y){
        // This function is used to change the scene
        if (app_state == 0) do:
                app_state = handleMenuKey(key);
        else if (app_state == 1) do:
                app_state = handleBattingKey(key);
        else if (app_state == 2) do:
                app_state = handleRunsKey(key);
}
```

handleRunsKey takes in the current key pressed and return the new state value accordingly with making appropriate changes.

When the application is executed first, the user can see a menu screen with few texts drawn. These are the details of the project member and also the title of the project is written. To do these in OpenGL, we have a method *glutBitmapCharacter()* which takes in the font and a single character as parameters. This method will draw the given character at a position specified in the position of *glRasterPos*. The following logic is used to display any string on the screen:

```
function drawString(string, x, y, z, font){
        // This function is used to draw a string
        setRasterPosition(x,y,z);
        for each character c in string do:
                glutBitmapCharacter(font, c);
        return;
}
```

On drawing a single character, the *RasterPostion* is incremented based on the size of the drawn character in terms of pixel along the *y-axis*. It should be noted that the entire rendering is taking place on a screen of *800*x*800* pixels. Any changes to the screen are being managed by the *reshape* method which will make sure to keep the entire screen at the center of the desktop.

Upon resizing, the method defined in *glutReshapeFunc* is called internally by the program and also makes sure to pass in the new changed width and height into this function. The function is defined to identify the aspect ratio factor and multiply this to the screen coordinates in such a way that the new *800*x*800* screen is always situated at the center of the application window.

In the Menu screen, we have also added a random text color effect for one of the string. This has been achieved by drawing the string with random colors after a certain duration. Initially, we draw a string and start time by using the *clock()* method

to capture the current time. Then we perform a check at every interval and see if the time has crossed some seconds. At this point, we draw the same string in different colors. The entire logic for this is given below:

```
function drawChangeColor(string, x, y, z, font){
        // This function is used to draw a string with colours.
        if (timer == 0) then do:
                drawString(string, x, y, z, font);
                r,g,b = getRandomColors();
                startClock();
                timer = 1;
        else if 0.75s has passed after timer started then do:
                timer = 0;
                startClock();
                drawString(string, x, y, z, font);
        else do:
                drawString(string, x, y, z, font);
}
```

On Pressing ‘a’ or ‘A’ on the keyboard the application changes to the next scene where it shows a cricket field, bowler, two batsmen, and stage. Each of these components is rendered separately with each being called in a particular order. In

```
function drawBattingScene(){
        // This function is used to draw a batting scene.
        drawGround();
        draw6Line();
        drawStadium();
        drawPit();
        drawWhiteCrease();
        drawWicket ();
        drawSecondBatsman();
        drawWicket();
        drawBowler();
        drawBatter();
}
```

general, when the application is in batting state the following logic of code is being called by the *display* method as shown before. Each of the function calls is defined to draw corresponding components and called in a particular order so that the correct component is rendered first. Following components are used in drawing a scene in the Batting state:

- The *drawGround()* function calls in a logic to render a rectangular green shape from the coordinates *(0,200)* to *(800,0)* as its top-left and bottom-right coordinates.
- The function *draw6Line()* draws a straight line of width 1 pixel from *(0,195)* to *(800,195)* coordinates in white color which represents the boundary line for the game to hit 4 or 6.
- The function *drawStadium()* is used to draw the background stadium which consists of a stage with each row of people sitting and also a scoreboard showing the current score. The lights on top of the stadium are drawn by using a rectangular mesh.
- The function *drawPit()* is used to render the pitch area where the bowler would throw the ball. In order to produce a realistic field, we used the same color as of the original field and then using a polygon drawing method defined in OpenGL we rendered the pit. The shape of the pit resembles that of a parallelogram.
- The function *drawWhiteCrease()* is used to draw the white line behind which the bowler should bowl or the batter should reach while taking the run. This is again being drawn similar to that of a boundary line but is drawn at 2 positions. One of them is drawn from the points *(70,160)* to *(60,125)* and other from the points *(640,160)* to *(630,125)*.
- The method *drawSecondBatsman()* draws the second batsman who takes a single run. Its implementation is similar to that of *drawBatsman()*.

- The function *drawWicket()* is used to draw the wicket. Here it takes in 2 parameters showing the location to draw the wicket at. Each of the components in a wicket is drawn by using lines of larger width. The following is the logic used:

```
function drawWicket(x,y){
        // This function is used to draw a wicket
        setLineWidth(3.9);
        drawLine(50+x, y+148, x+50, y+215);
        drawLine(58+x, y+148+5, x+58, y+215+5);
        drawLine(66+x, y+148+10, x+66, y+215+10);
        drawLine(50+x, y+125, x+66, y+215+10);
}
```

- The method *drawBatter()* is used to draw the batter in the scene. We designed by using a simple stickman figure who is holding a bat. The entire logic for drawing the batsman is defined here:

```
function drawBatter(x,y){
        // This function is used to draw a batsman
        checkBallIncoming();
        drawCircle(x+40, 117+y, 25);
        drawTorso(x,y);
        drawArms(x, y, rot);
        if (taking_run == 1) then do:
                leg_animation(x, y);
        else do:
                drawLegs(x, y);
        checkBallHit();
}
```

The function *drawCircle()* is used to draw the head and the method *checkBallHit* and *checkBallIncoming* are used to check if the bat hits the ball and the ball is incoming to the batsman or not. Based on these methods, the values in the different methods are modified. The *rot* variable in the

drawArms() method is used to specify the angle of rotation so that the batsman can hit the ball by swinging his bat.

- The function *drawBowler()* is used to draw the bowler. The component of the bowler is similar to that of the batsman but with slight modification in the arms as they would be holding the ball. The following pseudocode is used to draw the Bowler:

```
function drawBowler(x,y){
        // This function is used to draw the bowler
        drawCircle(x+40, 117+y, 25);
        drawTorso(x,y);
        if (key_press == 1) then do:
                leg_animation();
                rot += rot_factor;
        else do:
                drawLegs();
        drawArms(x, y, rot);
}
```

The functions make use of simple shapes to draw any component. For example, to draw the lightings in the stadium, we create a rectangular mesh which is a composition of a large number of small rectangles arranged side by side. Similarly, for other objects like a bat, we made use of Polygon to draw the shape at fixed coordinates and rectangles to draw stadium, wickets, net and so on. For performing any transformation on these objects, we are changing the coordinates at which they are drawn.

In order to perform the running animation, we need to move the legs from extreme left to right and vice versa. We use a variable *rotateLegs* to keep track of the angle between the left and right leg. To move the position of the legs, we use 2 variables *leftLeg* and *rightLeg* which hold the offset values for drawing the legs. The code given in the next section is used for this operation.

```
function leg_animation(){
        // This logic is used to animate the running scene
        x_speed -= factor;
        if (rotateLegs <= lowlimit) then do:
                leftLeg += factor;
                rightLeg -= factor;
        else if (rotateLegs < uplimit) then do:
                leftLeg -= factor;
                rightLeg += factor;
        else if (rotateLegs >= uplimit) then do:
                leftLeg += factor;
                rightLeg -= factor;
        rotateLegs += 1;
}
```

As each of the operations for movement is being done, the *x-axis* coordinates are changed by some factor to move the position of the component left or right and the *y-axis* is being changed to move the components on up and down axis.

The ball is being drawn as a red color filled circle and it has its own properties like *xposition*, *yposition*, and *radius*. To provide movement to the ball, we add offset values to the variables mentioned before. The ball is in one of the two states: thrown, idle. This information is needed to provide the exact movements for the ball. Since the ball will be in the hands of the bowler, it is necessary to throw the ball at the proper angle of the hands.

The goal of this project is to present a realistic scenario of different actions taking place in the game of cricket. All the animation and actions shown in the project are constrained by the rules of the game. Consider the below scenario:

When the batter is taking a run, it is needed for the ball to be returned after the completion of the r, so we used a variable to keep track of this and only return the ball when it is needed. Also for taking the 4 or 6, the ball has to be thrown out of the scene and then transited to the part where it actually hit or crosses the boundary. To do this transition we are changing the state of the app using the variable *app_state*

and showing the proper animation at that state. The following code maintains the ball in the second state of the application:

```
function drawBall(x, y, r){
    // This logic is used to render the ball
    If (isThrown==0) do:
        If (armAngle <250) do:
            drawCircle(x, y, r);
        else if (armAngle >250) do:
            throwBall();
        else if (taking_run==0)
            drawCircle(x, y, r);
    else do:
        drawCircle(x, y, r);
        x += ball_x_speed;
        y += ball_y_speed;
        // bouncing the ball
        if (y<=150) do:
            ball_y_speed = 2;
        else if (y>=offsety) do:
            offsety -= 2;
            ball_y_speed = -10;
        if (x>=800) do:
            app_state = 2;
            if (key == '6') do:
                y = 600 + randOffset;
            else if (key == '4') do:
                y = 380 + randOffset;
            else if (key=='1') do:
                app_state = 1;
                take_run = 1;
            else  if (key=='0') do:
                app_state = 1, call out_();
    }
```

The above actions are performed in state 1 and can be sent to either state 2 or remain in state 1 itself.

The second scene contains a stadium, a line, and a ball that comes flying from the top left corner. The ball can either go across the boundary and score a 6 or bounce once and take run 4. To draw all the components of this scene we have the following logic:

```
function drawScene2(){
        // This logic is used draw the second state screen
        drawGround();
        drawStadium();
        drawCurveLine();
        drawBall();
        moveBall();
}
```

The functions *drawGround, drawBall,* and *drawStadium* are similar to that which was defined for the first scene before. The method *drawCurveLine()* creates a curved boundary by using the concept of Bezier Curves. The following implementation shows how this is created.

```
function drawCurveLine(){
        // This logic is used draw the curve line
        for (t = 0 to segments; t++) do:
                tval = t/segments;
                x = getBezierPoint(650,750,750,650,tval);
                y = getBezierPoint(0,75,125,200,tval);
                drawPoint(x,y);
}
```

The above method has the component segment which is used to produce a smooth curve. The larger the value the smoother is the curve. The method *getBezierPoint* returns a value which is obtained by passing each of the coordinates of the control points and its corresponding parameter value. The values that passed to the *getBezierPoint* are used to align the shape of the curve, in our case we wanted the curve to represent the boundary line of the cricket as seen from the side view.

The implementation for getting the new curve point using the Bezier Function is defined below:

```
function getBezierPoint(a, b, c, d, t){
        // This logic is used get coordinates for the curve line
        C = (1-t)*(1-t)*(1-t);
        S = 3*(1-t)*(1-t)*t;
        T = 3*t*t*(1-t);
        F = t*t*t;
        return C*a + S*b +T*c + F* d;
}
```

To draw a circle of some radius on the screen we use the parametric equations to draw the *x* and *y* coordinates. The following implementation logic is used for this case:

```
function drawCircle(x, y, r){
        // This logic is used to draw the circle
        segments = 100;
        twicePI = 2*3.142;
        for (i=0 to i<=segments;i++) do:
                x = x + (r*cos(i*twicePI/segments));
                y = y + (r*sin(i*twicePI/segments));
                drawPoint(x,y);
}
```

To draw a point after applying the rotation, we make use of the general pivot point rotation equations to get the *x*, *y* coordinates. The following algorithm is used to return the new coordinate values after rotating about some pivot point.

```
function rotatePoint(x, y, px, py, theta){
        theta = theta * 3.142 / 180;
        x = px*(1-cos(theta)) + py*sin(theta) + cos(theta)*x - sin(theta)*y;
        y = py*(1-cos(theta)) - px*sin(theta) + sin(theta)*x + cos(theta)*y;
}
```

Usage

The application starts with showing the menu screen where each of the team member's names and USN details are shown. To navigate to the main screen, you can press the key ‘A’ or ‘a’ from the keyboard. Now you are shown with a scene that consists of a batman and a bowler in a cricket field. The following keys can be used for further interactions in the main menu:

- Key ‘4’: On pressing this key a scene showing the batsman taking a 4 is played.
- Key ‘6’: On pressing this key a scene showing the batsman taking a 6 is played.
- Key ‘1’: On pressing this key a scene showing 2 batsmen taking single run is played.
- Key ‘0’: On pressing this key, the next ball will hit the wicket and dismiss the batsman.
- Key ‘R’: After the game is over, you can restart the game by pressing this key.
- Key ‘A’: When the batsman is out and to continue the game, we press this key.

Results

In this project, we have a set of animations or screens that are implemented. The below is the list of all of them.

1. Home Screen:

   This shows the name of our project, it also shows the names of the various team members.

2. Stadium Screen:

   This shows the batsman, as well as the bowler, are standing at the respective positions. It also shows the stadium behind with the crowd cheering. We have also implemented floodlights and the scoreboard as well.

3. Score Management System:

   This system is responsible for setting the target score and managing each of the runs taken by the batsmen. It also shows the game end screen to the user.

4. Animation for hitting a 6:

   Once the user clicks on the number 6 on the keyboard, the following animation is shown on the screen. Firstly, the bowler bowls and then the batsman hits it for a six.

5. Animation for hitting a 4:

   Once the user clicks on the number 4 on the keyboard, the following animation is shown on the screen. Firstly, the bowler bowls and then the batsman hits it for a four.

6. Animation for scoring a run:

   Once the user clicks on the number 1 on the keyboard, the bowler would start to throw the ball towards the batsman and the batsman would hit it, after this both the batsman would run and score a point.

7. Animation for wicket hit:

   Once the user clicks on the number 0 on the keyboard, the bowler throws the ball and it hits the wickets of the batter side. The batting team then would lose a wicket and when all the 10 wickets are down before scoring the target, the game is a tie.

Conclusion

We have successfully implemented Cricket Simulation using OpenGL. We have also added quite a few animations relating to Cricket. Using this project, we have demonstrated the various parts that are involved in a game of Cricket. We have implemented animation that can be helpful for a new learner to quickly learn the basics of the game in a short amount of time.

References

Following websites and blogs were referred for the OpenGL documentation and implementations:

1. Opengl.org
2. Opengl-tutorial.org

Appendix

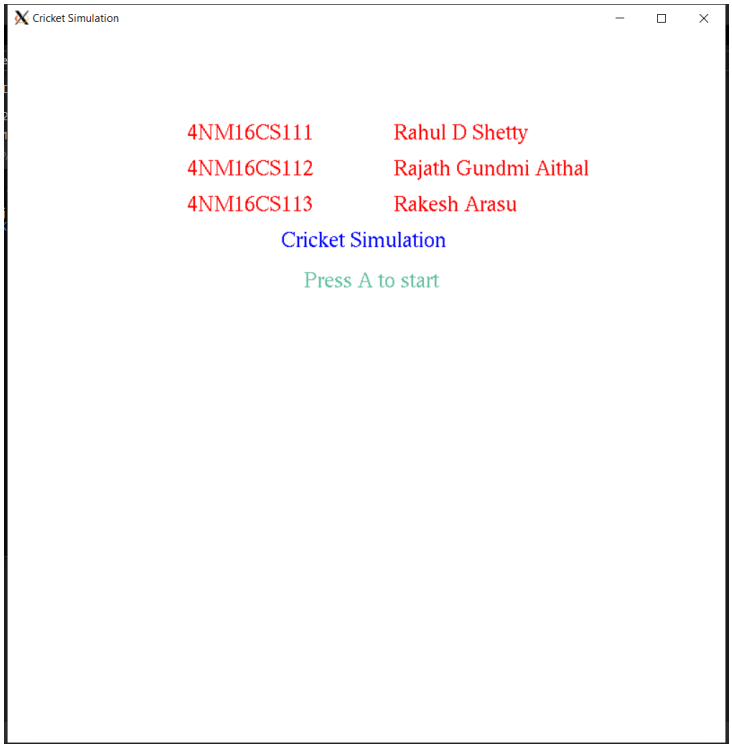

Figure 1:Main Menu

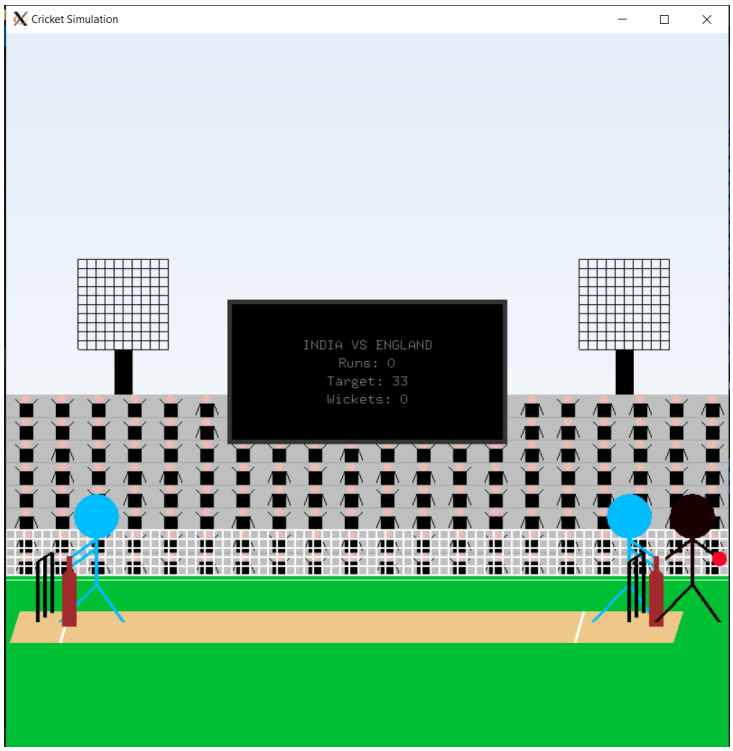

Figure 2:Batting Scene

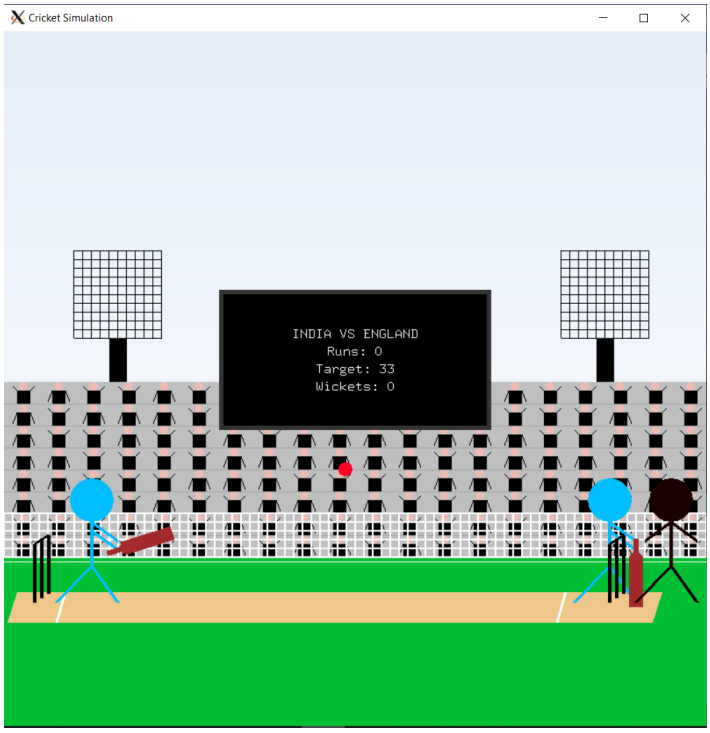

Figure 3:Batting Animation

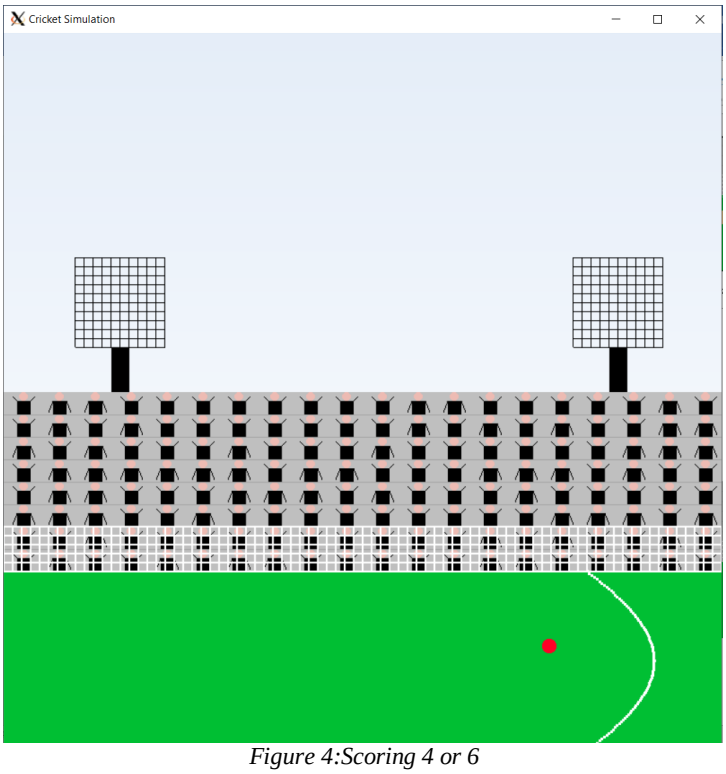

Figure 4:Scoring 4 or 6

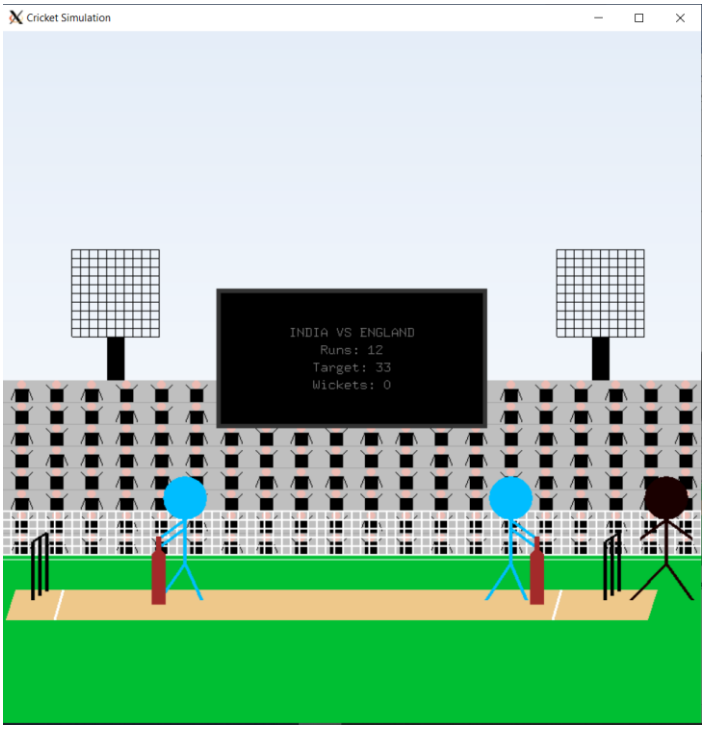

Figure 5:Taking a Single Run

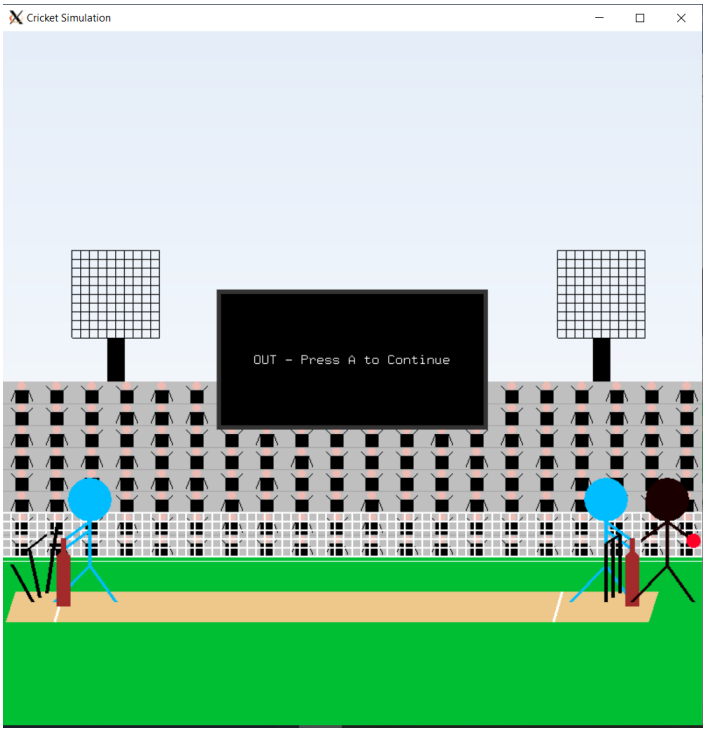

Figure 6:Wicket Hit