Project: ITC's Brick Slayer

Sibt ul Hussain

November 12, 2013

Deadline: Sunday 1st Decemeber, 2013 before 23h30

Attention

- Make sure that you read and understand each and every instruction. If you have any questions or comments you are encouraged to discuss your problems with your colleagues (and instructors) on Piazza.
- Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded (straight zero in the project — which can eventually result in your failure) and appropriate action as recommended by the Disciplinary Committee (DC can even award a straight F in the subject) will be taken.
- Try to understand and do the project yourself even if you are not able to complete the project. Note that you will be mainly awarded on your effort not on the basis whether you have completed the project or not.
- Divide and conquer: since you have three weeks so you are recommended to divide the complete task in manageable subtasks. We recommend to complete the drawing of bricks, board and ball in the first week, animations of balls and board in the second week and collision resolution by the end of third week.

Goals: In this project you will build a simple 2D game (ITC's Brick Slayer – see Figure 1 and video "game.avi") using the techniques learned during the course. More specifically, this project has two major goals: firstly to consolidate the things you have learned during the course. Secondly to introduce you complete program development cycle using existing libraries (OpenGL in this case) since you will be using libraries to build a real system.

1 Instructions

We provide complete skeleton (skeleton code draws three shapes using triangles) of project with detailed instructions and documentation. In other words all you need to know for building the game is provided. Your main task will be to understand main parts of skeleton and fill in the missing parts (or add new parts) to complete the game. However, before proceeding with code writing you will need to install some required libraries.

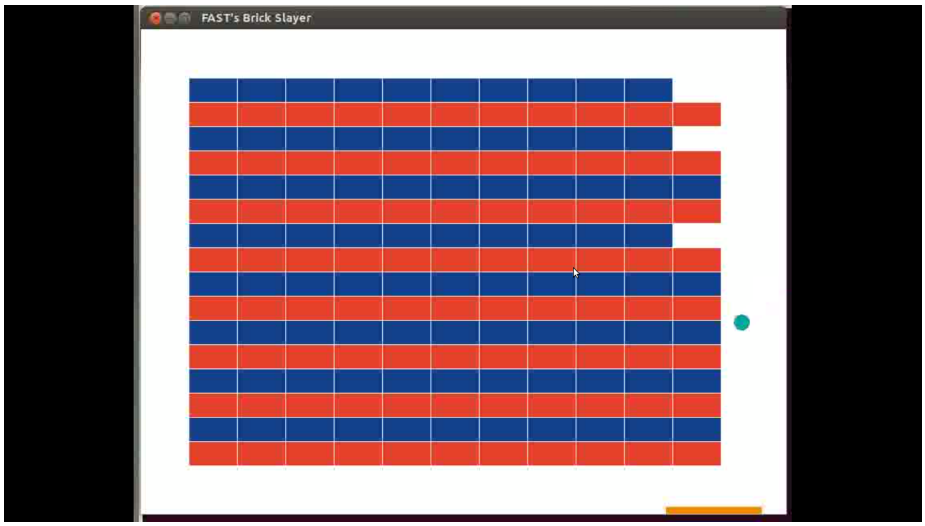

Figure 1: A screen shot of the game. For more detailed information please see the video "game.avi".

1.1 Installing libraries on Linux (Ubuntu)

You can install libraries either from the Ubuntu software center or from command line. We recommend command line and provide the file "install-libraries.sh" to automate the complete installation procedure. To install libraries:

1. Simply run the terminal and go to directory which contains the file downloaded file "install-libraries.sh".
2. Run the command

```
bash install-libraries.sh
```

3. Provide the password and wait for the libraries to be installed. If you get an error that libglew1.6-dev cannot be found, try installing an older version, such as libglew1.5-dev by issuing following on command line

```
sudo apt-get install libglew1.5-dev
```

4. If you have any other flavour of Linux. You can follow similar procedure to install "OpenGL" libraries.

1.2 Compiling and Executing

To compile the game (skeleton) each time you will be using "g++". However to automate the compilation and linking process we use a program "make". Make takes as an input a file containing the names of files to compile and libraries to link. This file is named as "Makefile" in the game folder and contains the detail of all the libraries that game uses and need to linked.

So each time you need to compile and link your program (game) you will be simply calling the "make" utility in the game directory on the terminal to perform the compilation and linking.

```
make
```

That's it if there are no errors you will have your game executable (on running you will see three shapes on your screen). Otherwise try to remove the pointed syntax errors and repeat the make procedure.

1.3 Drawing Shapes

Since we will be building 2D games, our first step towards building any game will be to define a canvas (our 2D world or 2D coordinate space in number of horizontal and vertical pixels) for drawing the game objects (in our case ball, board and bricks). For defining the canvas size you will be using (calling) the function "SetCanvas" (see below) and providing two parameters to set the drawing-world width and height in pixels.

```
/* Function sets canvas size (drawing area) in pixels...
 *  that is what dimensions (x and y) your game will have
 *  Note that the bottom-left coordinate has value (0,0)
 *  and top-right coordinate has value (width-1,height-1).
 *  To draw any object you will need to specify its location
 * */
void SetCanvasSize(int width, int height)
```

Once we have defined the canvas our next goal will be to draw the game objects using basic drawing primitives. For drawing each object we will need to specify its constituent point's locations (x & y coordinates) in 2D canvas space and its size. You will be using only triangles as drawing primitives to draw all shapes. For this purpose, skeleton code already include a function for drawing a triangle (see below) at specified location. Recall that a triangle has three vertices (points) so to draw a triangle we will need to provide these vertices (points) locations along with triangle color. Note that each color is combinations of three individual components red, green and blue, so to set or change a color of any object or background you will need to pass these components value.

```
/* To draw a triangle we need three vertices with each
 *  vertex having 2-coordinates [x, y] and a color for the
 *  triangle.
 *  This function takes 4 arguments first three arguments
 *  (3 vertices + 1 color) to draw the triangle with the
 *  given color.
 * */
void DrawTriangle(int x1, int y1, int x2, int y2, int x3,
          int y3,float color[]/*three
                component color vector*/)
```

Remember that you can do your drawing only in the `Display()` function, that is only those objects will be drawn on the canvas that are mentioned inside

the Display function. This `Display` function is automatically called by the graphics library whenever the contents of the canvas (window) will need to be drawn *i.e.* when the window is initially opened, and likely when the window is raised above other windows and previously obscured areas are exposed, or when `glutPostRedisplay()` is explicitly called.

In short, `Display` function is called automatically by the library and all the things inside it are drawn. However whenever you need to redraw the canvas you can explicitly call the Display() function by calling the function `glutPostRedisplay()`. *For instance, you will calling the `Display` function whenever you wanted to animate (move) your objects; where first you will set the new positions of your objects and then call the `glutPostRedisplay()` to redraw the objects at their new positions. Also see the documentation of `Timer` function.*

To complete the drawing phase of your game you will have to define two major functions `DrawRectangle()` (to draw bricks and board) and `DrawCircle` (to draw ball). Both of these shapes will be drawn using triangles primitives. Now you have to think how to draw a rectangle and circle using triangles. Recall a rectangle in 2D can be specified by two vertices representing its lower-left-corner and top-right-corner – see Figure 2), so your DrawRectangle must take as input position of two vertices and rectangle colour and must draw the rectangle (using triangles) at specified position. In comparison, for drawing a circle we need to know its center point (a single vertex (x_c, y_c)) and its radius. For drawing circle you will need to use multiple triangles and consequently to draw these triangles you will need to the location of vertices. [Hint: From trigonometry, we know that position of any point P_c on the circle boundary can be give by this simple relation, *i.e.* $P_c = (r\cos\ theta, r\sin\theta)$].

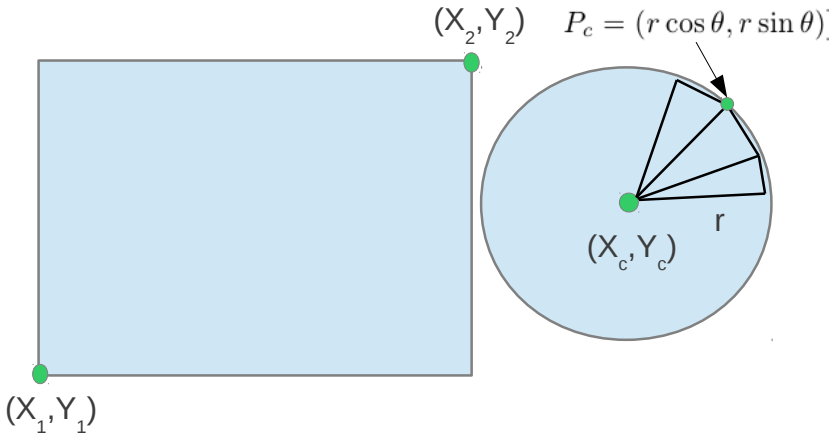

Figure 2: Representation of a rectangle and a circle. A rectangle can be represented using two vertices (given by (x_1, y_1) and (x_2, y_2)) representing its lower-left-corner and top-right-corner. A circle can be represented by its center point (x_c, y_c) and radius r.

Note that when drawing a particular object you will need to record its position, size and other necessary properties, *e.g.* for each brick you will need to record its visibility property (*i.e.* whether a brick is visible or not). Similarly you will need to record the displacement properties of ball (how much to move ball in x and y directions in each second) and board.

1.4 Interaction With Game

For the interaction with your game you will be using arrow keys on your keyboard (you can use mouse and other keys as well). Each key on your keyboard have associated ASCII code. You will be making using of these ASCII codes to check which key is pressed and will take appropriate action corresponding the pushed key. *E.g.* to move the board right you will check for the pressed key if the pressed key is left arrow you will move the board left (change its position). Keyboard keys are divided in two main groups: printable characters (such as a, b, tab, *etc.*) and non-printable ones (such as arrow keys, ctrl, *etc.*). Graphics library will call your corresponding registered functions whenever any printable and non-printable key from your keyboard is pressed. In the skeleton code we have registered two different functions (see below) to graphics library. These two functions are called whenever either a printable or non-printable ASCII key is pressed (see the skeleton for complete documentation). Your main tasks here will be add all the necessary functionality needed to make the game work.

```
1  /*This function is called (automatically by library)
2  * whenever any non-printable key (such as up-arrow,
3  * down-arraw) is pressed from the keyboard
4  *
5  * You will have to add the necessary code here
6  * when the arrow keys are pressed or any other key
7  * is pressed...
8  * This function has three argument variable key contains
9  * the ASCII of the key pressed, while x and y tells the
10 * program coordinates of mouse pointer when key was
11 * pressed.
12 * */
13 void NonPrintableKeys(int key, int x, int y)
14
15 /* This function is called (automatically by library)
16 * whenever any printable key (such as x,b, enter, etc.)
17 * is pressed from the keyboard
18 * This function has three argument variable key contains
19 * the ASCII of the key pressed, while x and y tells the
20 * program coordinates of mouse pointer when key was
21 * pressed.
22 * */
23 void PrintableKeys(unsigned char key, int x, int y)
```

1.5 Collision Detection

Finally, once you have done the drawing and animation of objects on your canvas. Your final goal will be to detect collisions (collision test) between objects and take necessary actions, *e.g.* to check whether ball and a brick are colliding or not, if yes than render the brick invisible. Collision detection can be simple as well as complex. In this game, you will be first implementing a very simple procedure for finding collision between two objects. Once done, you can improve your collision detection algorithm to make your game more realistic as in video.

The simplest way to find collision between a sphere (ball) and a rectangle (board or bricks) consists of two steps. In the first step we find the enclosing square of the ball – How ?. Now once we have found the bounding square of the ball, the problem of collision detection between ball and board (or bricks) can be treated as finding collision between two rectangles.

1.5.1 Collision Test

Finding collision between rectangles is extremely simple. Given two rectangles A and B defined by their centers and half-extents (half width, half height) – *c.f.* Figure 3, their overlap can be determined quite simply by checking whether both the rectangles are overlapping in both x and y dimensions or not *i.e.* $overlap = D.x < 0 \&\& D.y < 0$, where

$$\begin{aligned} \mathbf{D} &= |centerB - centerA| - (halfExtentsA + halfExtentsB) \\ &\text{Specifically,} \\ D.x &= |centerB.x - centerA.x| - (halfWidthA + halfWidthB) \\ D.y &= |centerB.y - centerA.y| - (halfHeightA + halfHeightB) \end{aligned} \tag{1}$$

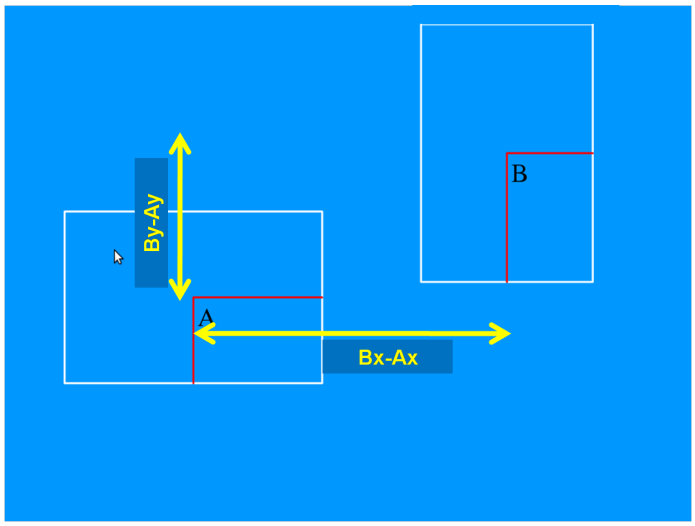

Figure 3: Illustration of two rectangles.

1.5.2 Penetration Vector

Collision test only tells us whether two objects are colliding are not, however in different scenarios we are required to resolve the collision *i.e.* move the objects such that their boundaries are only touching each other. So in addition of collision test, we look for a penetration vector which can tell us the level of penetration and direction of penetration of the object. Thus given a penetration vector $\mathbf{P}$, a colliding object can be moved in the direction opposite to $\mathbf{P}$ by its magnitude to resolve the collision.

Once we have found that there is a collision, we can resolve it using penetration vector $\mathbf{P}$. $\mathbf{P}$ direction $\Theta_{\mathbf{P}}$ will be the axis where there will be minimum overlap and the magnitude $||\mathbf{P}||$ will be simply the value of overlap *i.e.* $||\mathbf{P}|| = \min(D.x, D.y)$. So we will move the object by $||\mathbf{P}||$ in the direction -$\Theta_{\mathbf{P}}$ to resolve the collision.