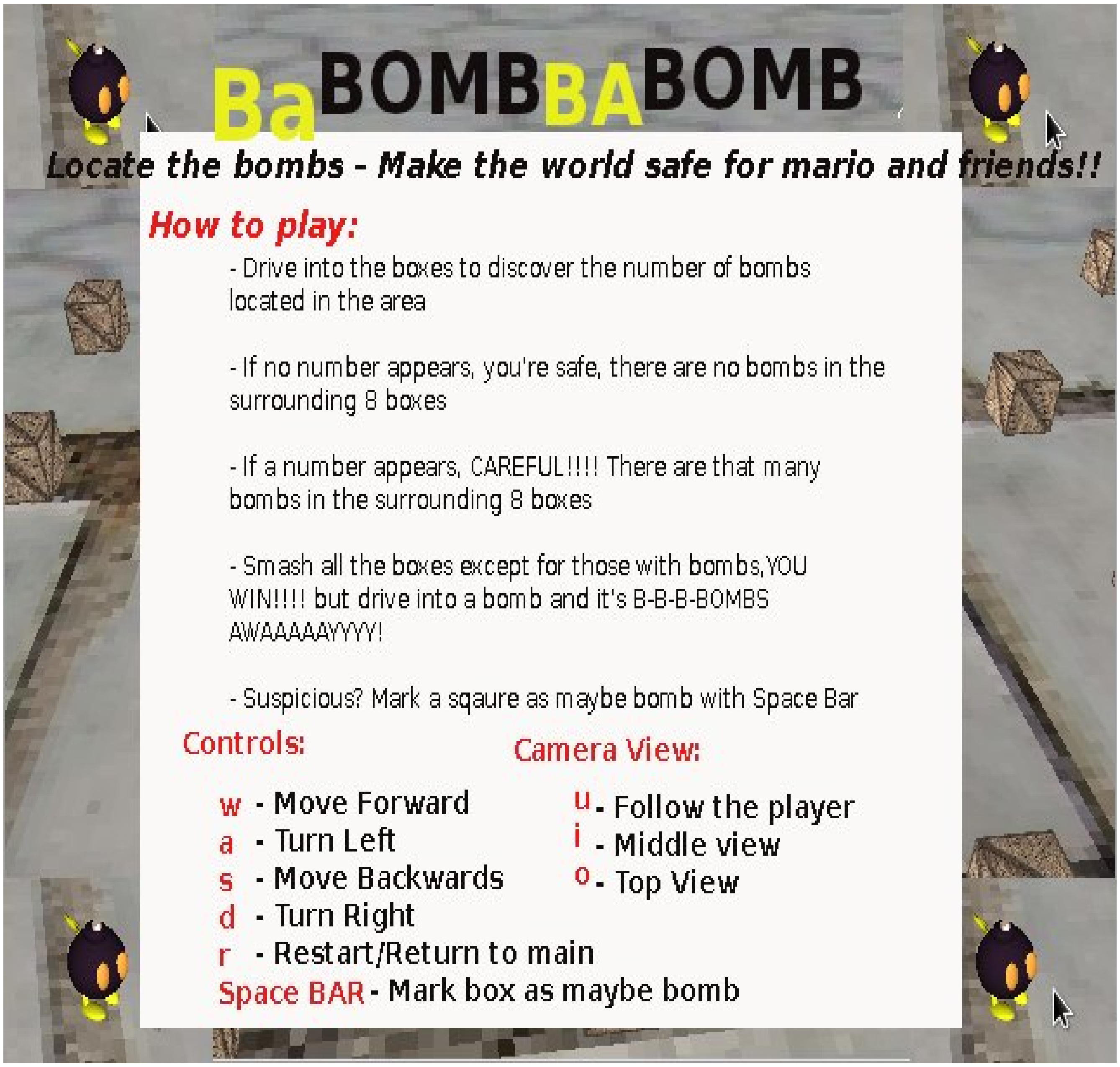
Game Project

Patrick Monahan - 08661413



BaBOMBBABOMB

Locate the bombs - Make the world safe for mario and friends!!

How to play:

- Drive into the boxes to discover the number of bombs located in the area

- If no number appears, you're safe, there are no bombs in the surrounding 8 boxes

- If a number appears, CAREFUL!!!! There are that many bombs in the surrounding 8 boxes

- Smash all the boxes except for those with bombs,YOU WIN!!!! but drive into a bomb and it's B-B-B-BOMBS AWAAAAAYYYY!

- Suspicious? Mark a sqaure as maybe bomb with Space Bar

Controls:

w - Move Forward
a - Turn Left
s - Move Backwards
d - Turn Right
r - Restart/Return to main
Space BAR - Mark box as maybe bomb

Camera View:

u - Follow the player
i - Middle view
o - Top View

The aim of this project was to create a 3-D game using OpenGL. To meet the necessary requirements I chose to make a mine-sweeper style game, I felt this gave me the opportunity to implement many of the project requirements such as lighting, texturing, user interaction, collision detection, winning/losing(Although it is much more common to lose than win in this game) as well as allowing me to explore simple animations within the game-play.

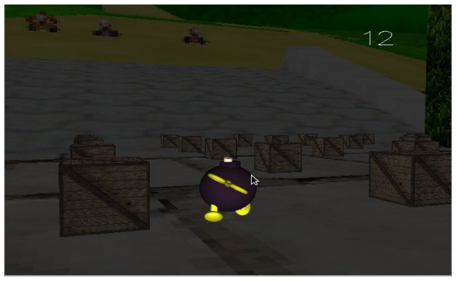
User Movement:

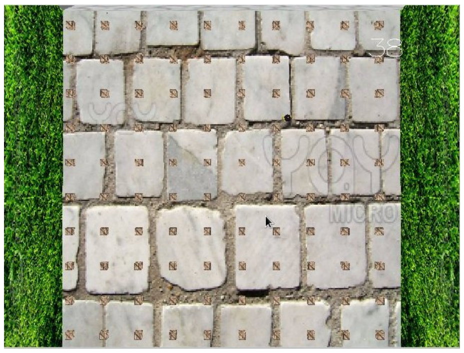
The first aspect of the game I implemented was the movement of the user. I worked with the teddy bear model given in lab 4 at first and made sure the movement was correct before proceeding with other game aspects. The player object is given a position and direction vector to track the where-abouts and movement direction of the player, the direction vector is initially {0,0,-1}, pointing down the direction of the z-axis, when the player turns, I rotate this vector by the angle of rotation, and ensure the vector is kept normalised, this allows me to keep track of the "forward" direction of the player. When the player moves forwards, the position vector is incremented by twice the direction vector, on the next render the player is then translated into the new position, to move backwards the direction vector is subtracted from the position vector, this ensures smooth movement in the desired direction.

Camera:

Chase camera:

After dealing with player movement I decided to implement a "chase" camera, which can be set in game play by pressing "u", where the camera follows the player as they move around the game. The camera is set using the gluLookAt() function, the position is set by subtracting 10*(players direction vector) from the players position, this sets the camera behind the player and follows them as they move. The position which the camera is looking at is then set using the player position vector.

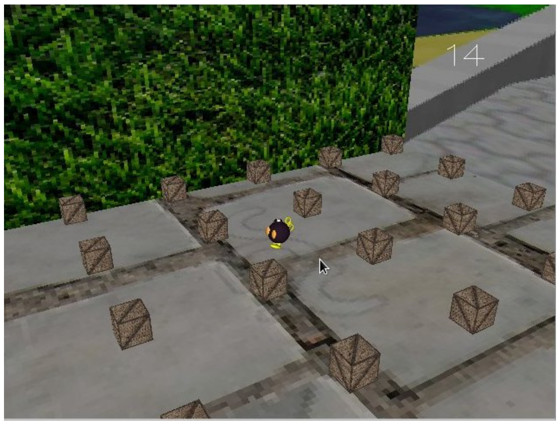




Top View:

I decided to allow different camera views to suit the players preference, I implemented a top view,accessed by pressing "i" while in game play, that simple places the camera in the middle of the board in terms of the x and z axes and raised 120 in the y axis, it then looks at 50 in the x-axis and -50 in the z-axis, ie. the middle of the board.

Middle View:



The “middle” view,accessed by pressing “o”, is a camera view where the camera position is static in the center of the board but follows the player around the board. The gluLookAt() is set at position (50,20,-50) and is passed to the player position vector, which determines the direction in which it looks.

Grid:

The game grid is created using a 2 dimensional STL vector of square objects. The grid is a 100x100 area over the x and z axes split up into a hundred individual squares, each square, a 10x10 space, contains a square model that is placed in the middle of the square. A square object contains all necessary information about each individual square, this includes among lots of other information, the model type for that square, the value(which is the number of bombs in the surrounding 8 squares), whether the square has been pressed or not, whether the square has been marked as “maybe a bomb”, whether the player is on this particular square and also the actual position of the model in the 3D world.

When the player “smashes” the first box, the grid is generated with the number of bombs specified in the grid class, bombs are placed at random around the board, but they can not be place on the players current position. This makes sure you don’t die on your first move, that wouldn’t be very fun at all! There is then another pass through the 2-D vector which looks for bombs, if bombs are found it then increments the 8 squares around the bomb. This updates each square so that the value of each square is the number of bombs in the surrounding 8 squares.

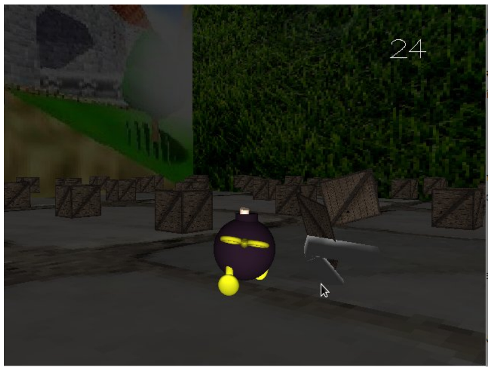
As the player moves around the board the part of the grid they are on is updated so PlayerOn = true, this is very useful for when it comes to doing collision detection.

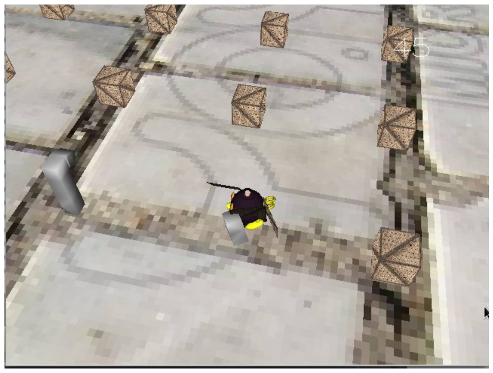
Collision detection:

To make the game more fun to play, I used collision detection to smash the boxes rather than having the player press a button. I used a method for collision detection based on the Euclidean distance between the player and th box object. This was made quite simple and more efficient as I keep track of the players position in the game space as well as keeping track of which particular square of the grid the player is one.

On each render of the scene, the players coordinates are passed in to a grid::smashBox method in the grid class, this calculates which square the player is currently on and calls the square::smashBox method, for that particular square, passing in the coordinated of the player. The square::smashBox method checks whether the box has already been smashed, if it has, nothing is done, if the box is still intact and the player is within the coordinates of the box model, then the box is smashed, playing the box smashed animation, and square::pressed is set to true.

The box smashed animation sets a counter, and while this counter is greater than 0 it rotates all the sides of the box object, as the box is rotating the number model is drawn and when the box has finished exploding, a model of a number is left representing the number of bombs in the adjacent squares.





There is also bounds checking to ensure that the player cannot go outside the game space into the 3-dimensional abyss.

Lighting

I wanted to create an almost cartoon-like feel to this game, as is quite evident with some of my blatant super Mario knock-offs. I thought an ambient lighting would be the simplest and best option to help achieve this. I used glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientColor); to set the style of lighting. This illuminates everything in my scene, allows for shading but creates no shadows behind objects.

Texturing:

For my texturing I used bitmap images for my textures, I found a bit map loader online which loads in a bit map image, I load in the bitmap images and bind them to a GLuint to be used as textures in my game. I could then map textures onto polygons using glTexCoord(). Everything in my game space is textured using bitmapped images such as the floor, boxes and walls.

To create my boxes, the best solution was to create a cube made up of squares, created using glVertex3f(), by creating seperate squares for each side I could then create the animating of the swirling squares when the player collides with a box. I found a suitable texure for a box online and mapped it onto the squares.

The walls are textured with screen shots from Mario kart and Mario 64, as this was the style of game I was going for, and also I thought it was fun.

BA-BOMB-MAN / Hierarchical object:

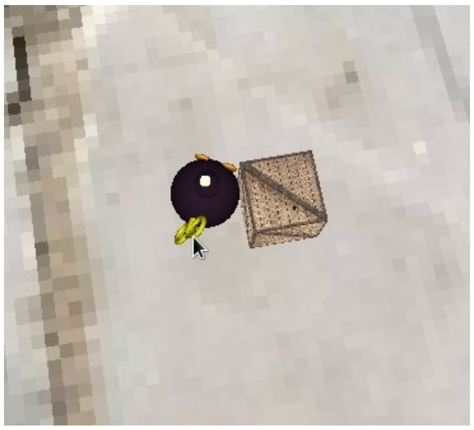
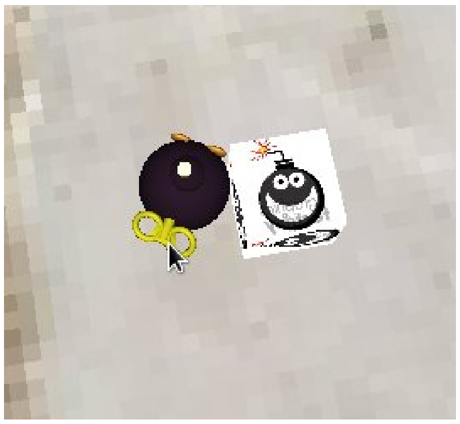
The player in my game is ba-bomb-man, he was modelled in the image of a super-mario b-bomb, I created him as a number of separate models created in blender. I created a main bomb model which was his body, then a wick, eyes and feet models. I decided to color him in openGL rather than applying a texture, I initially intended to texture him but when I tested out some colorings I thought it looked well and suited the feel of the game.

I also created a “winder” which is placed coming out of his back, it rotates about it’s own axis independently of the movement of the whole bomb object.

I created an impression of walking by rotating the feet as you walk. When methods are called to move the player forward, backwards or to turn, a walking angle is incremented, it is incremented to a max value and then decremented to a minimum value. Ba-bomb-man’s feet are then rotated accordingly, in the opposite direction of each other to give the impression of a walking bomb.

Maybe Bomb:

When a square is suspected of being a bomb, a player can press the space bar when they are close to it to change the texture and mark it as a possible bomb, a new texture is bound before the box is drawn and this can be used as a warning for later in the game, it can also be unset by pressing space bar again.

Menu:

BaBombBaBomb has a basic menu on start-up, it is a simple textured rectangle. When you press on the “find the bombs” the game is initialised and you can play the game. When credits is pressed the texture is simply changed to show a credits page with some very basic info. You can also the exit the game from the menu by pressing the exit button.

Loser cut scene:

If you should so unfortunately crash into a bomb, it's GAME OVER! The camera is set to a front view of the player, all player controls are disabled and baBomb man is flipped into the air. This is done quite simply using a series of rotates and translates to simulate the bomb being flipped into the air by a bomb exploding!!

A message is also written to screen to inform the user of their inability to successfully complete the game.

Issues:

Rotation Angle:

When implementing the direction vector, I had trouble when implementing the vector rotation rule:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta.$$

The vector length was not being correctly maintained and strange figures were being calculated, I could not find a proper solution for this that allowed me to continue to rotate the current direction vector. The solution I came up with was to create the direction vector by rotating the original direction vector by a rotation angle that is maintained for the player. Each time the player turns a new rotation vector is calculated by using the rotation angle to rotate the original player's direction vector using the above formula.

Loading models:

An issue arose when I was loading in models that I had created in Blender. Whenever I created a model that was anything more than a simple cube created in blender the model would not load into the 3D world space. I realised after quite sometime this was because I had not placed materials on the object in blender as I had planned to texture my exported objects in openlGL, openGL then read the 3DS models as transparent and did not draw them in my 3D world space.