Programming for 3D Coursework
Interactive 3D Game Development in XNA
Candidate Number 77409:
Debris Destroyer 3

The main motivation of this game was to create an advanced modification of the original "FuelCell" game provided by MSDN in their XNA game development tutorials. I wanted to modify the game so that it would use controls similar to that of a traditional first/third person PC shooter (i.e., using the WASD keys for movement along the X and Z axis, the Space and Shift keys for the Y axis, and using the mouse for aiming and firing).

The aim of the game is to destroy the floating pieces of debris in the landfill before the ship's fuel is fully depleted. The player is able to replenish the ship's fuel by collecting the fuel cells along the floor of the landfills. If the player is able to destroy all the pieces of debris before the fuel runs out, that level is complete and the game moves on to the next level where the ship's fuel depletes even quicker. Points (or money as the game's story describes it as) are given for every piece of debris that is destroyed, and are accumulated until the ship's fuel is empty and the player loses the game.

DD3Game Class

This is the main class which is responsible for the running of the entire simulation, handling all the objects within it, loading all the graphics and models. Most of the game's logic is done within this class.

Methods:

Initialize:
This method instantiates the some of the objects necessary for the game to start running.

LoadContent:
This method is responsible for loading all the images, sounds, models and other dynamic content in the game.

PlaceFuelCellsAndDebris:
This gives all the fuel cells and debris randomly assigned positions on the game's map.

GenerateRandomPosition:
This method gets called by PlaceFuelCellsAndDebris in order to generate random positions along the floor for the fuel cells.

GenerateRandomPosition2:
This is similar to the method above but it generates random positions for the debris that gets placed floating in the atmosphere.

IsOccupied:
This is used by the GenerateRandomPosition methods to ensure that the position being generated isn't already occupied by another object.

Update:
This is the method that gets called upon every tick of the game's cycle. Most of the user's input is listened for here, and this is also where most of the calculations take place. First, it waits for the user to transition between the starting screens, and then it runs the game. This method automatically reduces the "fuel" of the ship and either waits for it to reach 0, or for the player to have destroyed all the debris. When one of these criterions have been met, the game changes state to either the "loss" screen, or the "win" screen respectively.

Collision detections are also kept track of here. Whenever the ship collides with a fuel cell, the fuel meter is updated, and when a bullet collides with a piece of debris, the score is incremented.

Shoot:
This method is called by the Update method. It waits for the mouse's left button to be pressed, to create a new object of the "Bullet" class and place it in the atmosphere.

Reset:
This method is called when either the player wins or loses the round, and chooses to play again. The ship's position is reset to the centre of the floor, and re-assigns all the debris and fuel cells with new positions, and clears the stage of any bullets that may have still been flying around.

<u>GameObject Class</u>
This class represents all of the objects within the game. Everything inherits its initial variables and methods, that required by all the objects in order for the game to function properly and efficiently.

CalculateBoundingSphere
This method calculates the radius of the object's bounding sphere relative to the model.

DrawBoundingSphere
This method was used during the testing of the game, to show the object's bounding sphere while the game was running.

InitializeGameField
This assigns each piece of debris with one of three randomly selected models and places them onto the field.

Draw:
This method contains a switch case that selects which screen to render according the game's state.

DrawTerrain:
This method renders the game's ground model onto the screen

DrawStartScreen:
Draws the startup image into the screen

DrawInstructionScreen:
Draws the instruction image onto the screen

DrawControlsScreen:
Draws the controls image onto the screen

DrawWinScreen:
Displays a splash screen which includes text saying which level was completed, how much money has been collected so far, and instructions on how to continue.

DrawLossScreen
Displays a splash screen which includes text saying which level was failed, how much money was collected in total, and instructions on how to restart the game.

DrawGamePlayScreen
This renders the actual game onto the screen according to the camera's view matrix and projection matrix.

DrawStats:
During gameplay this method renders the text on either corner of the screen, including the "fuel remaining" and the amount of money earned.

Ship Class
This class represents the spaceship in the game, that the player controls. It inherits all the initial variables and methods of the GameObject class.

LoadContent:
The model of the ship is loaded here, and so is the bounding sphere that is used to detect collisions. The bounding sphere's radius is also calculated and applied here.

Reset:
This is called by the DD3Game class when the game is reset, to set ship back to its original position, and face it in its original direction.

Draw:
This is method renders the ship's graphics on the screen, and is responsible for drawing the rotation of the ship depending on its variables which get changed throughout the duration of the

game.

Update:
This is where the ship listens for input from the user (via keyboard and mouse) and changes its position and direction accordingly.

ValidateMovement:
This method called by the Update method, and checks whether or not the future position of the ship (depending on the user's input) is a valid one. If the ship is going to collide with debris, or is going to move outside the limited bounds of play, it will stop moving.

CheckForDebrisCollision:
This is called by the ValidateMovement method, to see if the ship collides with debris.

CheckForFuelCollision:
This is called by the ValidateMovement method, to see if the ship collides with a fuel cell. If it does, it sets that fuel cell's boolean Retrieved variable to true.

<u>Debris Class</u>
This class represents the floating pieces of debris in the game's environment. It inherits all the initial variables and methods of the GameObject class. This class also has a LoadContent method to load the models and bounding spheres, and a draw method to render the objects onto the screen if they are not destroyed.

<u>FuelCell Class</u>
This class represents the fuel cells lying across the ground of each level. This class is similar to the Debris class, but will only render on the screen if they have not been retrieved.

<u>Bullet Class</u>
This class represents the bullets that get fired out of the ship when the player clicks the mouse button. It takes the position and direction of the ship upon creation, so that it knows where to start from, and where to move to. The model used by this class is one I created in Autodesk 3DS Max 2011, which consists of a simple sphere shape, with a standard self-illumination (light-green colour).

Update
This method is responsible for calculating the future position of the bullet and its bounding sphere (dependant on the ship's position and direction), and updating it accordingly.

CheckForCollision
This checks to see if the bullet collides with any debris, and if it does, it sets that piece of debris' Destroyed variable to 'true'.

Camera Class
This class represents the game's camera, which picks up all the objects within range to render them onto the screen.

Update
This method updates the position and orientation of the camera according to the ship's yaw and pitch, so that it remains constant at a fixed position behind it, no matter which way it's facing.

GameConstants Class
This class is just a list of static variables that remain constant throughout the duration of the game. Several references will be made to this class, in order to get certain pieces of data.

Advanced Functionality
The main challenge of the implementation of this game, was to get the ship to aim and fire according to user's mouse input, as there are several complex mathematical calculations regarding matrices and vectors.

Firstly, the game waits for any change in direction of the mouse, and modifies the ForwardDirection (for horizontal rotation) and AimDirection (for vertical rotation) float variables according to the difference in the mouse's previous and new position.These two variables are then used to create two rotation matrices (one for the X axis and one for the Y axis) which are then multiplied together to get the proper full rotation of the ship. The product of these two matrices is then multiplied by a translation matrix, which uses the ship's Position vector the move it around the map.

The ForwardDirection and AimDirection variables are then passed onto the Camera's Update method, which uses these variables for the rotation matrix. This rotation matrix is modified by a "Matrix.CreateFromYawPitchRoll" method, which takes the ForwardDirection value to determine the yaw of the camera, the AimDirection value, for the pitch of the camera, and a value of 0 for the roll of the camera (as it did not need to rotation along the Z axis).

Another major challenge was getting the bullets to fly along the direction that the ship was facing. The bullets also take the direction variables from the ship, but use them in a slightly different way. A vector is created that updates the bullets position along the Z axis, and uses the ship's AimDirection variable to create a rotation along the X axis and then the ForwardDirection variable for it's Y axis upon every call of the bullet's Update method. The bullet's position is then calculated adding this vector to the ship's position (i.e., the starting position of the bullet).

The further challenge arose when trying to prevent the cursor from going off the screen, which would result in the user being brought out of the game if they were to try and click outside the area of the game's screen. In order to prevent this from happening, I had to change the code so that the mouse's position was set to the middle of the screen upon every update, and changes in the direction of the ship were calculated by the distance of the mouse's new position away from the center of the screen.

The fuel bar at the top of the screen was also quite tricky to implement. It works by taking two images, one of the grey and red empty bar, and one of the green block it's filled up by. The first one is drawn onto the screen as normal, but it's the green bar that is drawn of top the red one that changes width according to how much fuel is left. In the spritebatch.Draw method, the size of the rectangle is specified, with the width of it being set as the amount of fuel remaining divided by 5 (because the maximum amount is 2500, and the length of the bar is 500pixels).

Other minor pieces of functionality that were implemented include:

- A simple level system, that makes the ship's fuel decrease faster and faster as the game progressed.
- Additional GameState enums, to add extra splash screens before the game starts.
- Sound effects that play at given moments in the game.
- Added simple pausing functionality [1]

References:
[0]FuelCell Game code: http://msdn.microsoft.com/en-us/library/dd940288.aspx
Date accessed: 20/04/12
[1] How to Pause: http://msdn.microsoft.com/en-us/library/bb195026(v=xnagamestudio.31).aspx
Date accessed: 22/04/12
[2] Floor and blocks textures: http://image.shutterstock.com/display_pic_with_logo/81383/81383,1199311592,6/stock-photo-computer-chip-closeup-texture-or-background-8192686.jpg
Date accessed: 22/04/12
[3] Keyboard Image: http://www.saurdo.com/images/keyboard.png
Date accessed: 24/04/12
[4] Oil Barrel texture: http://spiralforums.biz/uploads/post-4216-1195750822.jpg
Date accessed 24/04/12
[5] Lazer sound effect: http://www.freesound.org/people/THE_bizniss/sounds/39459/
Date accessed: 24/04/12
[6] Power Up sound effect: http://www.freesound.org/people/StudioCopsey/sounds/77245/
Date accessed: 24/04/12
[7] Explosion sound effect: http://www.freesound.org/people/inferno/sounds/18400/
Date accessed 24/04/12
[8]Laser sound effect 2: http://www.freesound.org/people/aust_paul/sounds/30935/
Date accessed 24/04/12
[9]computer chip texture: http://image.shutterstock.com/display_pic_with_logo/81383/81383,1199311592,6/stock-photo-computer-chip-closeup-texture-or-background-8192686.jpg
Date accessed 24/04/12