COMPSCI 777 S2 C – Assignment 1

Due Date: Monday 9th August, 10.30pm

Computer Science

This assignment is worth 6.67% of your final mark for COMPSCI 777 S2 C.

Assignment Description

Screenshot of a solution for assignment 1

This assignment requires you to develop and implements a simple adventure game in which a player moves around a 3D scene and catches creatures. In the game the player's view position motion is controlled by key presses that move the player at a constant velocity (forward and back plus left and right shift using the 4 arrow keys), and the player's view orientation is controlled by the mouse. Forward motion (up arrow key press) will be in the direction the player is looking. The game scene at the start of the game consists of a texture mapped ground plane and objects (in the screenshot on the right I use billboard trees). Every few seconds a new 3D model of a creature (I use rabbits for this demo) automatically appears on the ground plane at a random position. After a creature has entered the game it moves at a constant velocity to a new position every frame, though creatures must avoid colliding with other creatures or objects and they must not move of the grid. If the bounding box of a creature hits the bounding box of an object its starts eating the object which is simulated by decreasing the size of the object until it disappears. If an object has been completely eaten points are deducted from the player and the creature continues to move.
The goal of the game is to save the objects from getting eaten and to catch as many creatures as possible. The player may do this by moving close to a creature and capturing it with a key press. The player may also use key presses to increase or decrease his/her velocity. Points are scored by successfully capturing a creature ("hit"), but points are deducted for a "miss".
During the game the current value of several variables is displayed (in text within the game window): frame rate, count of # of objects currently in the scene, count of # of creatures currently in the scene, game score, and time remaining.
Each game runs for a fixed number of seconds. The game terminates early if all objects have been eaten. When the game is over, the game scene is frozen, the input controls are disabled, and the game status values are left displayed in the window.

This game uses no complexity management for displaying objects or for determining whether objects might collide. Objects should be stored in a linked list, and processing them (collision checking and drawing) will be done by linear list traversal. Thus, all objects are drawn even if they are outside the field of view (they will be clipped by the OpenGL rendering pipeline). Checking for object collisions will be an $O(N^2)$ complexity operation (for N = # of 3D objects). Thus, if the player is slow at picking up objects, more and more objects will be created and the frame rate will get slower. A slower frame rate makes motion control for the player more difficult, so poor play is penalized and leads to low scores. But good play removes objects from the scene and maintains a high frame rate which leads to high scores.

Learning Objectives

- Learn to program a Windows application which uses OpenGL.
- Learn to implement camera motion control that has constant velocity independent of rendering rates.
- Learn to implement object collision detection using axis aligned bounding boxes (AABB).
- Learn to implement text display by drawing bit mapped font characters.

Problem Specifications

Here are some more detailed specifications for the game:

1. You must use a **texture mapped square ground plane** which is in the *xz*-plane (i.e. the plane defined by the equation *y*=0). You are allowed to use the grass plane from last semester's 372 assignment. If you are motivated please feel free to create a more interesting "indoor" scene with walls.

2. Your **creatures** must be shaded 3D models. You can choose any creature model you want as long as it is reasonable complex. You are welcome do download models from the web and you can choose any format you like. You can use the classes from last semester's 372 assignment to define light sources.
    - Each creature has a **current position**, **orientation** and **velocity**. When the game starts there are no creatures in the scene. Creatures will be placed one at a time at intervals of *I* seconds at random positions on the ground plane.
    - If a creature's bounding box is about to go of the grid the movement direction of the creature must be reflected on the grid boundary.
    - If the bounding boxes of two creatures collide their direction is reversed.
    - If a creature is hit by the player (see (5) for more details) it is removed from the game.
    - The game keeps track of the current number of creatures on the ground.

3. Your **objects** (which are eaten by the creatures) must be 3D models or billboards of a complex object. You are allowed to use the billboard trees from last year's 372 assignment.
    - If an object's bounding box is hit by a creature the creature stops and starts eating the object which is simulated by decreasing the object's size linearly.
    - When an object shrinks its bounding box should shrink, too. In this case the creature will continue to move in the next time step and either hit again the shrunk bounding box or continue on its original path.
    - If the object size has reached zero it is removed from the object list and the score is decreased by K points (you may choose K).
    - The game keeps track of the current number of objects on the ground.

4. The **game player** has a current position and orientation. The game camera is attached to the player so the player and camera will always have the same position and orientation used to control the camera view. Therefore, this game is a "first person shooter" type of game. The player is not visible on the screen and hence there is no geometry for it which needs to be rendered.
    - The player camera may move as follows:
        i. Mouse motion controls the look direction yaw and pitch (but not roll).
        ii. Arrow keys control forward and back motion (up/down arrow) and side to side strafing (left/right arrow). The player's direction of forward motion is the same as the current camera view orientation (as set by the mouse controls).
        iii. The player may not move below the ground plane (y=0), but the player may move to any x or z position and to any y >= 0.

iv. The speed of motion (player/camera velocity) is initially set at a default value, but the speed may be increased or decreased by 10% with each press of the + and - keys. Note: the player should use the numeric + and - keys so he/she doesn't have to press the SHIFT key, however some laptops do not have a numeric keypad, so alternate keys may have to be used.

5. The player can capture creatures on the ground plane. When the 'g' key ("get") is pressed the player may catch a creature if and only if the following conditions are true:

   - The "hit point", the (x,y,z) point that is distance DELTA from the player's current (x,y,z) position (i.e., directly in front of the player at a distance DELTA) falls within the creature's bounding box. The creature's bounding box is the axis aligned box formed by the bounding planes that are at the minimum and maximum x, y, and z coordinates of the object. The value of DELTA must be 2 times the diameter of your 3D creature's bounding box. The diameter of a bounding box is defined as the maximum of the length, width and height of the bounding box.

   - If these conditions are true, i.e., the player hit the creature, the creature is removed from the scene, the count of "creatures on the ground" is decreased by 1, and the game score is increased by H points (you may choose the value for H).

   - If these conditions are false, i.e., the player missed the creature, the score is decreased by M points (you may choose M).

6. **BONUS:** Do something else cool!! The maximum possible bonus is 5% of the total mark for this assignment.

Design and Implementation Requirements & Hints

Here are some instructions and hints to help you with completing this assignment. In order to get started do the following:

1) First read through some (or all) of the tutorials on the 707 web pages. You should definitely read the Win32 tutorials #1 Creating a Window (First Win32 Program), #2 DeviceContext, #3 ColorText, and #8 FullScreen and both of the OpenGL tutorials #1 Triangle and #2 FramesPerSecond. Read as many of the other tutorials about handling input (especially keyboard input) as you wish. These web pages are copies of the source code written by GameTutorials.com (Ben Humphrey) and you may download all of them from the GameTutorials web site, http://www.gametutorials.com/Tutorials/tutorials.htm.

2) Create your own source files that include functions that do the following:

   - Define the main function WinMain [This function is created automatically if you use .NET, though it is named _tWinMain]. This main function is REQUIRED and it must have EXACTLY the same prototype as shown in the tutorials. Most of WinMain should be nearly exactly the same as in the tutorials.

     - NOTE: When you create a project in a .NET solution or a Visual Studio Workspace, you should create a new project of type "Win32 Project" (not a "Win32 Console Project").

   - Define the callback function WndProc [This function is created automatically if you use .NET]. This function is also REQUIRED and it must have EXACTLY the same prototype as shown in the tutorials. You may modify its code body as needed. This is where all Windows System messages (such as window create, paint, destroy, etc.) MUST be

handled and it is also one of the places you may place code that handles user inputs (keyboard and mouse).

- **Initialize OpenGL**
  For the OpenGL functions, you must use the calls as shown in the two OpenGL tutorials. However, it is not necessary to decompose the various initialization steps into separate functions as the author did. You may combine them into one long WinMain function. Although this may seem counter to what you have learned in your software engineering classes, game program development often "breaks the rules" of good software design for the sake of execution speed. As a student in this class you'll need to learn the best trade off between developing fast, efficient code (which may not be easy to maintain) and using structured design and other software engineering principles that produce code that is easy to maintain but may be less efficient in its execution speed.

- Calculate and display the frame update rate. When you calculate the frame rate, do NOT display it as part of the window title bar because this slows down program execution. Instead, follow the example in the Win32 tutorial #3 ColorText and draw the frame rate as text within the window. In order to compute and display the frame rate you should download and read the following tutorials at GameTutorials.com:

  - OpenGL Tutorial #10, Time Based Movement, http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_Pg1.htm.

  - OpenGL tutorial #34 Fonts, http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_Pg4.htm

  The TimeBasedMovement tutorial builds upon previous ones in the OpenGL tutorials collection at GameDevelopers.com, so you may want to download and browse through some of them also (the 5 previous tutorials on Camera).

- You may copy and paste and modify some of the GameTutorials.com code when you do this assignment. The main goal is to make sure you understand the steps for initializing a window and handling keyboard inputs.

3) You are required to design a C++ class for your 3D creature. The class must have a constructor, destructor, a step (update) function, and a draw function. You may include other functions if needed.

   - A creature must have these data members:

     - (*x,y,z*) position, e.g., it's center point

     - (*x,y,z*) direction vector (*y* direction component must be zero)

     - a scalar velocity

     - a bounding box: min *x,y,z* and max *x,y,z*
       Hint: the bounding box coordinates could be relative to the object's position. If so, then the step function will not have to change the box coordinates every frame. It's best to make an independent class for the bounding box which you can then also use for the objects in the scene.

   - The constructor must initialize the position, direction, velocity, and bounding box subject to the constraints described above.

   - The step function receives one parameter, `int dt`, the elapsed frame time in milliseconds. It must update the position by converting `dt` into an elapsed time in seconds, then using that time value, the velocity, and the direction vector to update the position vector. However, it must check two constraints:

- If the new position would be off the grid, reflect the direction vector. Since the boundaries are axis aligned this is easily achieved by negating either the x- or the y-coordinate.

- If the new position would cause the creature's bounding box to intersect the bounding of any other creature, reverse the direction.

- If the new position would cause the creature's bounding box to intersect the bounding of an object then don't update the position and initiate the shrinking of the object. If the object has disappeared the position of the creature is updated again.

  - If you know about OpenGL display lists it would be wise to define one display list when the first creature is constructed, and then have all other instances of that creature use the same display list.

4) You may use the code in the GameTutorials.com example (CCamera class) for the **Camera Control**. This class includes 4 vectors (type CVector3), m_vPosition, m_vView, m_vUpVector, and m_vStrafe, but it doesn't include a view direction vector. The m_vPosition is the camera's "look from" point and m_vView is its "look at" point. Thus, the vector (m_vView - m_vPosition) is the (unnormalized) view direction vector. See the MoveCamera function of the CCamera class for an example of the use of these vectors.

   - The "hit point" for capturing objects should be the position vector plus DELTA times the normalized view direction vector.

   - Handle keyboard inputs that modify the camera view position. Your program must read some keyboard inputs, such as the up, down, left, and right arrow keys (or your favourite letter keys for direction control such as 'a', 's', 'w', 'x', etc.). Use the inputs from the keys to modify the look from x and z position of the gluLookAt function call (see the RenderScene function in file Main.cpp). The simplest way to check the input key value is to call the GetKeyState function. This is explained at the end of the Win32 Tutorial #5. The #defined values for the up, down, left, and right "virtual keys" are VK_UP, VK_DOWN, VK_LEFT, and VK_RIGHT. Thus, you may use,

```
if ( GetKeyState(VK_UP) & 0x80 )
```

5) **Bounding boxes**

   - You may wish to design a bounding box class (not required). A good reason to do so is that you can then design member functions such as, bool point_in_bbox(CVector3 v) that checks if the (x,y,z) point v is contained with this object's bounding box and bool bbox_intersection(boundBox bbox) that compares this object's bounding box with another.

   - You must figure out the algorithms for the point_in_bbox and bbox_intersection. The algorithms are very trivial, especially the point_in_bbox. Here is a helpful hint for the bbox_intersection algorithm: There are only 6 possible orderings of the values this.xmin, this.xmax, bbox.xmin, and bbox.xmax. 2 of those orderings are cases when the two bounding boxes cannot possibly intersect regardless of what the y and z min and max values are, and the other 4 are cases when the boxes may intersect depending on what the y and z values are. The same statement is true for this.ymin, this.ymax, bbox.ymin, and bbox.ymax and for this.zmin, this.zmax, bbox.zmin, and bbox.zmax. Thus, if either of the 2 cases is true for x or for y or for z, then the boxes don't intersect. Otherwise, they intersect.

☺ *Enjoy!*

Assignment submission

All files should include your name, UPI and ID in a comment at the beginning of the file. All your files should be able to be compiled under .NET without requiring any editing. In particular they should include the necessary OpenGL/GLU and Windows libraries.
Clearly indicate the source of any code you are using in your solution. All your files should include adequate documentation (this means enough documentation to enable me to understand what your are doing).

The assignment due date is Monday the 9th August 2004 (time: 10.30pm).
Please submit your assignments using the assignment dropbox. If the dropbox server is down please create a zip-file which contains all the files you want to submit and email it to me. The name of the zip-file must be "Ass1_<your_UPI>.zip" (e.g. Ass1_bwue001.zip).
Assignments submitted late get a 5% penalty. No assignments are accepted after the 10th August 2004, 10.30pm.

Please Submit:

- A .NET solution with all your source, header and resource files (e.g. texture images); but without the object code, i.e. without the 'Debug' and 'Release' directories which are created when compiling your code. Please put all files of your .NET solution into a zip-file.
- A text file ReadMe.txt (or a PDF file if you want to include images) with a description of any problems and any functionality not mentioned in the above specification (e.g. describe what you did for the bonus part).

Marking

Assignments are marked according to their technical quality, the game play experience, and according to how well they fulfil the given specification.