CSC 476 – Lab 1 due 10/2/08
A Simple Game

Please note that this lab should be completed **in pairs**! You may talk to one another about the lab, but you may not look at someone's working code other then your partner! If you can, I encourage you to pair with someone who took CSC 471 with me (as they will have a good code base to work with).

This assignment requires you to develop a game program that implements a simple adventure game in which a player moves around a 3D scene and collides with objects. In the game, key presses control the player's position and the player's view direction is controlled via the mouse. The game scene includes a wire frame grid ground plane. When the game starts the only 3D object that appears in the scene is the ground plane. But, every few seconds a new 3D mesh automatically appears on the ground plane at a random position. After an object has entered the game it moves at a constant velocity to a new position every frame, though objects must avoid colliding with other objects and they must not move off the grid. The goal of the game is for the player to collide with all the objects. He or she may do this by moving close to an object. Points are scored by successfully capturing an object ("hit").

Each game will run for a fixed number of seconds, and then the game is over. During the game the current value of several variables is displayed (in text within the game window): frame rate, count of # of 3D objects currently in the scene, count of # of objects encountered (i.e. collided with = game score), and time remaining.

This game uses no complexity management for displaying objects or for determining whether objects might collide. Objects will be stored in a linked list (or vector), 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). (Although you must use display lists to render your copies of the multiple game objects). 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. Limit the number of meshes that can enter the game to 10.

- **Learning Objectives**
  - Learn to implement a **game data structure** that adds and removes **3D objects** by storing them **in a linked list** or vector
  - Learn to implement **object updating** by **traversing the collection of game objects**
  - Learn to implement **object collision checking** by using **axis aligned bounding boxes (AABB)**
  - Learn to implement **camera motion control**

- Learn to implement **text display** in a drawing window by drawing **bit mapped font** characters

• Grading and Due Date

You must **demo your program in lab on 10/2/08.**

• Problem Specifications

Here are the **rules** of this **game** and **specifications** for your **implementation** of the game.

1. **Ground plane**

▪ The 3D scene contains a **ground plane** that is a wire frame grid in the X-Z plane (i.e., in the plane defined by the plane equation y = 0). You may modify the grid parameters as you wish, however, e.g., to make it larger. You may include any other scene elements you'd like (for example some simple hierarchical models in the scene) – if you do include scene elements you do not need to compute collision with them, but I encourage you to try.

2. **Game objects**

▪ Sitting on the ground plane will be **one or more 3D objects**. The size, structure, color, and other attributes of the objects will be determined by the game developer (you). With the exception that the object must be a mesh file (I recommend one of the smaller ones – bunny, gargoyle, or Cessna .These objects must be flat shaded.

▪ Each object has a **current position, orientation,** and **velocity**.

▪ When the game starts there is nothing in the scene except the ground plane (no objects). Each object will be placed one at a time onto the ground plane automatically every P seconds. You may choose the value for P, or you may handle it as an input value supplied at run time. Each new object must be initially positioned at a random X and Z coordinate, but with a Y coordinate such that the bottom of the object sits of the ground plane (Y=0). However, an object may not be placed off the edge of the grid and it may not be placed so that it overlaps with another object.

▪ Every frame each object moves in a constant velocity in its straight ahead direction to a new position. But, an object may not move off the edge of the grid, and an object may not move so that it collides with another object. If an object's bounding box is about to go off the grid, it must reverse its direction. If an object is about to collide with another object, it should just freeze and not move for that frame (note: this could lead to deadlock for two objects that are moving towards each other, but in this game you need not handle breaking such a deadlock). The game developer (you) may specify that all objects have the same velocity or that they have random velocities (within a minimum and maximum velocity range), and you may specify how the initial direction of an object's motion is determined.

▪ The goal of the game is for the "player" to collide with each object to make them stop moving. You may play with other effects when collision has happened but at very least the object must stop moving and have its color change.

▪ The game keeps track of the **current number of "objects on the ground"** and the **current number of "objects collided with".**

3. **Game Player**

- The **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" type of game. There is no geometry of the player that needs to be rendered.
- The **player** (camera) may **move** however you determine is best, but the user must be able to control:
- the **look direction** yaw and pitch (but not roll).
- **forward** and **back** motion ("w"/"s" keyss or **up/down arrow**) and **side to side strafing** ("a"/"d" keys or **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).
- 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.
- The **speed of motion** (player/camera **velocity**) is initially set at a default value, but you may add keys to increase the speed if desired..
- The **player's position and view orientation** can be **reset** to the initial values by pressing the **HOME** key.

4. **Game scoring**

- **H** points for every object **hit** (removed)
- The game keeps track of the **total number of seconds elapsed**. When the number of seconds reaches N, the game is over. You may choose the value for N, or you may design the game to input that value at run time.
- During the game these same statistics are displayed in the upper left corner of the game window

• Programming Design and Implementation Requirements

0. Write code for drawing the **ground plane grid** modified to be an appropriate extent and spacing for the mesh size (You may also make the ground plane a solid plane and texture mapped if you'd like).

1. **3D object class**

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

- An object 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 (or if desired you may store 'velocity' in the direction vector)
- an axis aligned bounding box (or AABB): 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.

- The **constructor** must initialize the position, direction, velocity, and bounding box subject to the constraints described above.
- The step function receives one parameter, dt, the elapsed time. It must update the position by converting dt into elapsed time in seconds, then using that time value, the velocity and direction, update the position. However, it must check two constraints:

1. If the new position would be off the grid, negate the direction vector and recompute the new position.
2. If the new position would cause the object's bounding box to intersect the bounding of any other object, do not update the position.

- The **draw** function should draw the object's geometry using OpenGL functions. You must use display lists to render your 3d objects. Please see the tutorial at: http://www.lighthouse3d.com/opengl/displaylists/index.php3?3

2. **3D objects collection**

- All objects must be stored in a **linked list or stl vector**. You may choose how to implement the linked list, either as simple pointer variables as one of the object's data members (e.g., a "next" variable), as a separate class, by using the C++ STL (Standard Template Library) linked list class, or other design of your choice. You may declare the head of the list as a global variable, or you may choose an alternate design.
- Drawing the entire scene should be done by traversing the objects in the linked list and invoking each object's draw function.
- Checking for collisions between objects when the step function updates an object's position should be done by traversing the linked list and comparing the object's bounding box to the bounding box of every other object (a second list traversal).
Note: this is an $O(N^2)$ computation. Later in this course you will learn algorithms to reduce the complexity of such a comparison.

Note that for grading, you can receive most of the points for this lab if we complete the above lab but with stationary object. Thus, I recommend that you start with game objects staying stationary. In other words, start by still reading in a .m mesh file and still compute its bounding box based on its random location in your world, but do not worry about making it move until you have a moving camera and collision detection working. Note that the mesh location should be accurate (i.e. somewhere sitting on your ground plane).

Do not focus a huge amount of energy on time elapse computation nor fps!
And put in “animation”/motion of the models second to last (put in time elapsed and fps very last!)