CSC 476 – Lab 1 due 1/19/17
A Simple Game

Mode: Please note that this lab can be completed **in pairs** – it is up to you.

Overview: This assignment requires you to develop a simple 3D game program that implements a simple adventure game in which a player moves around a 3D scene and collides with objects, collecting them. 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 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 object 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 collect all the objects by colliding with them. Points are scored by successfully capturing an object ("hit").

- **Learning Objectives**
  - Practice implementing a **game data structure** that adds and removes **3D objects** by storing them **in a linked list** or vector
  - Practice implementing **object updating** by **traversing the collection of game objects** using time based movement
  - Practice implementing **object collision checking** by using **axis aligned bounding boxes (AABB)** or bounding spheres (your choice)
  - Practice implementing **camera motion control** using time based movement
  - Practice keeping track of game data (and potentially practice **text display** in a drawing window by drawing **bit mapped font** characters (or other methods)) – data can be printed to the console for this program
  -
- **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 in the X-Z plane (i.e., in the plane defined by the plane equation y = 0). 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, during the game **one or more 3D objects which have a clear orientation (i.e. a front and back)** will appear over time. 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 (obj). These objects must be shaded! **Use vertex buffer objects or Vertex Array Objects to display of the game objects.**

- Each object has a **current position, orientation,** and **velocity** (this is a good time to start thinking about software design)
- When the game starts there are no meshes to collect. Each collectible mesh 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 either 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) or move in the opposite direction. 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. *Please consider spending some time tweaking variables so that movement is decent – playable*.
- 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 – ie there must be some visual side effect to 'show' the user they have collided with the mesh.
- The game keeps track of the **current number of "objects on the ground"** and the **current number of "objects collided with".** This data can be displayed to the console or screen

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" keys 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.
- Optional: consider adding a weak spring to your camera to help with more natural movement (camera lags behind the user slightly).

- All animations must use 'time based movement'. Search for current tutorials on **Time Based Movement**, (or to be even more pedantic.: http://gafferongames.com/game-physics/fix-your-timestep/ /).

During the game the current value of several variables is displayed (in text within the game window or to the console): frame rate, count of # of 3D objects currently in the scene, count of # of objects encountered (i.e. collided with = game score).

Note: This game uses no spatial dataset 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). 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 may get slower (depending on mesh complexity and number of objects). Potentially, slower frame rate makes motion control for the player more difficult, so poor play is penalized and leads to low scores. Limit the number of meshes that can enter the game to 10-50 (your choice depending on scaling – more meshes makes non-overlapping placement challenging depending on scale). Note that you must use Vertex Buffer Objects or Vertex Array Objects to represent and display the meshes.

In general, this lab is intended for you to consider how you might like to proto-type software design for your team game. Think about: how you might like to design a game object class, a camera class, a more sophisticated shader class (consider using how to handle uniform and attribute variables in general) and other aspects of your game. This is a chance to practice/experiment with various software design choices – your goal is for your team to choose the best "lab 1" to use as the start to your final game project. You can look at these articles on game programming patterns:
http://gameprogrammingpatterns.com/contents.html
And this opinion piece about OO vs. entity:
http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

- **Programming Design and Implementation** ***Information (if you want to propose an alternative reasonable design you can).***

0. **3D object class**

This is a good time to design a **C++ class** for your 3D object (I like most part of this short intro:
http://homes.cs.washington.edu/~tom/c++example/c++.pdf)

1. The class traditionally includes 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 or bounding sphere

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 bounding structures values appropriately or keep track of a transform matrix (and apply prior to collision)

- 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 vertex buffer objects or vertex array objects to draw the meshes.**
- Your mesh should be oriented correctly in the direction it is traveling

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.

Grading:

- Correct game camera: 15
- Accurate collisions and reasonable response: 25
- Mesh (time- based) movement:15
- Mesh orientation: 10
- Mesh shading and game look (is it primary colors? Too dark?): 20
- General game statistics & software design: 15