ECE 4122/6122 Final Project

(300 pts)

Due Date: Dec 2nd, 2025 by 11:59 PM

Submitting the Assignment:

All the files needed to compile your code including a CMakeLists.txt should be zipped up into a file called **FinalProject.zip** and uploaded to canvas. For the Final Project, you can choose between two types of submission:

1. You can develop your final project on your local machine. If you chose this type you will need to create a narrated video showing your program working and either upload it to canvas or provide a link its location. **You still need to submit your code** with the video.
2. You can develop your final project on pace-ice and just upload your code to canvas. You need to make sure it builds and runs on pace-ice.

Grading Rubric

AUTOMATIC GRADING POINT DEDUCTIONS :

Element	Percentage Deduction	Details
Does Not Compile (ignore if you provided a video)	40%	Code does not compile on PACE-ICE!
Does Not Execute as expected and/or Output is wrong format	10%-90%	The code compiles but does not produce correct outputs or does not execute as expected. UAVs flight paths are incorrect. UAVs shape/color is wrong.
Clear Self-Documenting Coding Styles	10%-25%	This can include incorrect indentation, using unclear variable names, unclear/missing comments, or compiling with warnings. (See Appendix A)

LATE POLICY

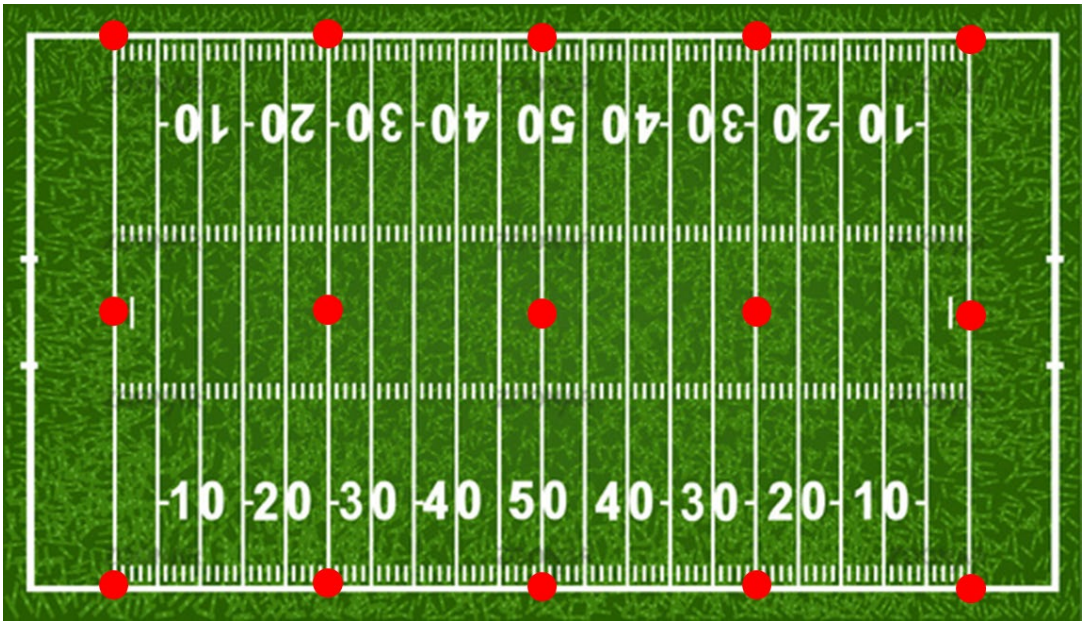
Element	Percentage Deduction	Details
NO late submission allowed		

GaTech Buzzy Bowl

The company you work for has been selected to develop a half-time show using UAVs. You need to develop a 3D simulation using std::thread and OpenGL to demo the show to the game organizers for their approval.

Below is a description of the show that you will be creating with a 3D simulation.

1) The show is made up of 15 UAVs that are placed on the football field at the 0, 25, 50, 25, 0 yard-lines as shown below by the red dots.



2) The UAVs remain on the ground for 5 seconds after the beginning of the simulation.

3) After the initial 5 seconds the UAVs then launch from the ground and go towards the point (0, 0, 50 m) above the ground with a maximum velocity of 2 m/s

4) As they approach the point, (0, 0, 50 m), they began to fly in random paths along the surface of a virtual sphere of radius 10 m while attempting to maintain a speed between 2 to 10 m/s.

5) The simulation ends once all of the UAV have come within 10 m of the point, (0, 0, 50 m), and the UAVs have flown along the surface for 60 seconds.

6) Each UAV has the following
    a. Each UAV has a mass of 1 kg and is able to generate a single force vector with a total maximum magnitude of 20 N in any direction.
    b. Scale each UAV so it is just small enough to fix in a 20-cm cube bounding box. Pick a UAV object from the attached zip file. There are lots of free 3D objects online and for an extra **20 extra bonus points** place a texture map on a 3D object of your choosing or on one of the included 3D objects.

7) You must develop a multithread application using 16 threads. The main thread is responsible for rendering the 3D scene with the 15 UAVs and a green (RGB=(0,255,0)) rectangle representing the football field in a 400 x 400 window. You will get **10 extra bonus points** for using a bitmap file called **ff.bmp** in the same location as the executable to apply a football field texture to the rectangle. The other 15 threads are each responsible for controlling the motion of a single UAV.

8) Create a class called ECE_UAV that has member variables to contain the mass, (x, y, z) position, (vx, vy, yz) velocities, and (ax, ay, az) accelerations of the UAV. The ECE_UAV class also contains a member variable of type std::thread. A member function start() causes the thread member variable to run an external function called threadFunction

```
void threadFunction(ECE_UAV* pUAV);
```

The threadFunction updates the kinematic information every 10 msec.

9) The main thread polls the UAV once every 30 msec to determine their current locations in order to update the 3D scene.

10) The coordinate system of the 3D simulation is defined as follows: The origin is located at the center of the football field at ground level. The positive z axis points straight up, and the x axis is along the width of the field and the y axis is along the length.

11) A camera location, orientation, and field of view should be used so that the whole football field and the virtual 10m sphere is in the view. You can display a semi-transparent representation of the virtual sphere

12) The flight of the UAV is controlled by the following kinematic rules
    a. The force vector created by the UAV plus the force of gravity (10 N in the negative z direction) determine the direction of acceleration for the UAV.
    b. Use Newton's 2nd Law to determine the acceleration of the UAV in each direction.

$$\vec{F} = m * \vec{a}$$

c. Use the equations of motion for constant acceleration in each direction (xyz) to determine the location and velocity of the UAV every 10 msec.

$$x = x_o + v_{xo}t + \frac{1}{2}a_x t^2$$
$$v_x = v_{xo} + a_x * t$$

d. You can use any method you want to maintain the UAVs flight path along the surface of the 10m radius virtual sphere. One possible method to consider is to PID controller to determine the force to apply to the UAV. See Appendix B for more details and you can use the attached PID_Sim.cpp for reference.

$$PID_{output} = (K_p \cdot \text{error}) + (K_i \cdot \text{integral}) + (K_d \cdot \text{derivative})$$

13) If the bounding boxes of two UAVs come within 1 cm of each other an elastic collision occurs. For simplicity we are going to model the UAVs as point objects and the UAVs will just swap velocity vectors for the next time step:

$$v_1 = v_2$$
$$v_2 = v_1$$

ECE6122 Students:

The magnitude of the color of your UAV should oscillate between full and half color throughout the simulation with a frequency of roughly 0.5 Hz.

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
```

```
    if (i < 5)
    {
        counter++;
        k -= i;
    }
    else
    {
        k +=1;
    }
    j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

```
/*
Author: <your name>
Class: ECE4122 or ECE6122
Last Date Modified: <date>

Description:

What is the purpose of this file?

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.

Appendix B: PID Controller

To use a PID (Proportional-Integral-Derivative) controller to keep a robot on a given path, you must continuously measure the robot's error relative to the path and use the three control components (**P**, **I**, and **D**) to calculate the appropriate steering correction. This process requires the UAV's position, a control loop to perform calculations, and force to implement the corrections.

Key concepts:

- **Setpoint:** The desired state for the UAV's . For path following, this is the UAV's 's ideal position and orientation on the path.
- **Process Variable:** The UAV's 's actual, measured position relative to the path, determined by its sensors.
- **Error:** The difference between the setpoint and the process variable. This is the main input for the PID calculation.
- **Control Loop:** A continuous process of measuring the error, calculating the PID output, and adjusting the UAV's 's force to correct its course.

Step 1: Determine the UAV's 's position and calculate the error

First, the UAV needs to determine its position relative to the target path. The error is a measurement of how far the UAV is from the path.

Step 2: Calculate the control signal

The PID algorithm calculates a corrective output by processing the current error, the sum of past errors, and the rate of change of the error. This output is then used to control the UAV's force.

The PID output is calculated using the following formula for each direction (x,y,z):

$$PID_{output} = (K_p \cdot \text{error}) + (K_i \cdot \text{integral}) + (K_d \cdot \text{derivative})$$

- **Proportional term (P):**
    - **Function:** Corrects based on the *current* error. The further the UAV's is from the path, the stronger the correction.
    - **Effect:** A large proportional gain (K_p) provides a rapid response but can cause the UAV to overcorrect and oscillate back and forth across the path. A small K_p results in a sluggish response. A typical range might be **10 to 100**. For a larger or faster UAV, this value could be much higher.
- **Integral term (I):**
    - **Function:** Corrects based on the *sum of past errors*. It addresses persistent, small errors that the proportional term might ignore. The value is often a fraction of K_p, perhaps in the range of **0.1 to 5**. Too high a value can cause slow, persistent oscillations.
    - **Effect:** Eliminates steady-state error, ensuring the UAV's eventually settles precisely on the path. Too large of an integral gain (K_i) can cause "integral windup," leading to overshoot.
- **Derivative term (D):**
    - **Function:** Corrects based on the *rate of change of the error*. It predicts future error by observing how quickly the UAV is deviating from the path. Its value is often in a range similar to (K_i) for example, **0.1 to 10**, though it can be higher on larger, slower systems.
    - **Effect:** Damps oscillations and stabilizes the system, allowing for faster and smoother movement without overshooting. Too large of a derivative gain (K_d) can make the system sensitive to noise.

Factors that influence PID gains:

Several factors critically impact the appropriate gain values for your specific system:

- **UAV Mass and Inertia:** A heavier drone will have more inertia, meaning it requires more force to accelerate and decelerate. This will generally require higher Kp and K_d values.
- **Propulsion System:** The maximum thrust, motor torque, and speed of the propellers affect how quickly the UAV can respond. A highly responsive propulsion system can often handle lower PID gains, while a less responsive one may require higher gains to compensate.

- **Response Time:** The desired responsiveness of the UAV dictates the tuning. A fast-moving racing drone will have very different gains than a large, slow-moving aerial photography platform.
- **Sampling Rate:** The frequency at which the PID loop runs affects the gain values. Faster sampling rates generally require lower gains, while slower rates may need higher gains to prevent instability.

How to approach tuning with typical ranges:

Since there is no universal "safe" range, follow a structured tuning process:

1. **Start Small:** Begin with very low Kp, and zero for Ki and Kd.
2. **Increase Kp:** Gradually increase Kp until the UAV starts to oscillate around the desired path. Then, reduce Kp to about half of this "ultimate gain".
3. **Increase Kd:** Increase Kd to dampen the oscillations and prevent overshoot. Add only as much as needed to avoid amplifying sensor noise.
4. **Increase Ki:** Finally, increase Ki to eliminate any small, persistent steady-state error. Do this slowly, as too high a value can cause slow oscillations.