**INFO6044 – Game Engine Frameworks & Patterns**

**Midterm Exam, 2022**

Instructor: Michael Feeney

## The exam format:

* You may use any resources you feel are necessary to complete the exam, but you are to answer the questions **on your own**. I will be looking for plagiarism (i.e. copying) very carefully. There is *no possible way* that the specific code to answer these questions, or the output to the screen, would be very similar to the look of another student’s code. Remember, this is a test and there are very clear policies about cheating on tests.   
  + <http://www.fanshawec.ca/admissions/registrars-office/policies/cheating-policy>
  + <http://www.fanshawec.ca/sites/default/files/assets/Ombuds/cheating_flowchart.pdf>
* It is an “open book” exam. You have access to anything you book or internet resource you’d like
* The questions are ***NOT*** of equal weight. The exam has **five (5)** questions and **nine (9)** pages.
* Your solution can be either graphical or console based (or graphical + console based if that’s helpful).
* **CLEARLY** indicate which answer goes to which question. My suggestion is that you place each answer in its own folder, named “Question\_01”, “Question\_02” and so on (or something equally clear). Another option is to create a Visual Studio solution and add a number of projects – one per question – to it. If I can’t make heads or tails of what question is what, I probably won’t even mark it.
* Do ***NOT*** do some clever “*oh, you just have to comment/uncomment this block of code*” nonsense. However, if the questions ***CLEARLY AND OBVIOUSLY*** build on each other, you may combine them (like if one question places objects, then the next one moves objects around with the keys) – even so, **MAKE IT 100% CLEAR** to me what questions the solution is attempting to answer.
* Place any written (“essay” or short answer) answers into a Word, RTF, or text file. Again, *clearly* indicate which question you are answering.
* If you are combining answers (which is likely), please indicate this with a “readme” file or some note (*not* buried in the source code somewhere).
* For applications: if it doesn’t build and run, *it’s like you didn’t answer it*. I’ll correct trivial, obvious problems (like you clearly missed a semicolon, etc.), but you need to be sure that it compiles and/or runs.
* You have until **11:59 PM** on **Friday, November 1th** to submit all your files to the appropriate drop box on Fanshawe Online.   
    
  **NOTE:** Although this may “look and feel” like a project, it isn’t, it’s an **exam**, so there is **no concept of “late marks**”; if you don’t submit your files the time the drop box closes, you don’t get any marks at all.

We need to pick a due date for this…

*Please don’t be late submitting.*

(Also be **SURE** that you are actually submitting the correct files)

* Your solution may **not** contain any third party “core C++” libraries (like boost). I will not have boost installed, and will not install it; as a result, if you using boost, your solution will *not* build, and you will receive a mark of zero (0).
* You many have other “utility” libraries, like ones to load textures, models, sounds, etc. However, make sure your submission is complete so that I can build your solution.
* When ready to submit, please delete all the “extra” Visual Studio files before zipping it up (remember this is C++, so all I really need is the .h and .cpp files, right?), like the “Debug” and “Release” folders with the “obj” files, as well as the intellisense file (in VS2017, that’s the “.vs” folder).
* **If the solution does not build (and run), I will not mark it** (so you will receive zero on questions that can't be built and/or won't run). When I say "run", I'm not speaking about some, random, unforeseen bug, but rather something that you should have obviously dealt with, like memory exceptions, etc.
* Unless otherwise indicated, all these solutions assume that you are creating/using a C++ project using Visual Studio 2008 through 2019 using the OpenGL 4.x API (with glfw, glad, and glm).

**“Robot battle arena!”**



You are going to simulate a futuristic robot battle, sort of like “battlebots” (Google that) only much, much more violent.

You will need two types of models:

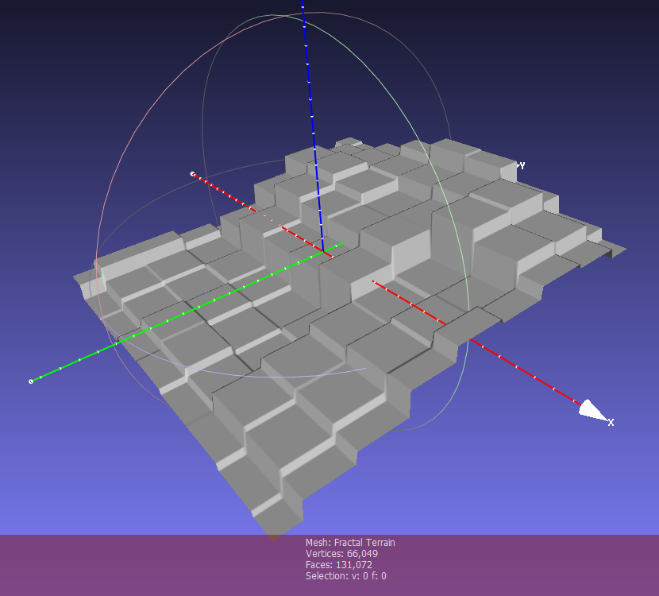
* A “multi-fractal” terrain (from mesh lab – more details later)
* Sphere, Bunny, and cube\_Low\_Res\_xyz\_n (or equivalent) models
  + Note:
    - the “bunny” will represent the robots
    - the “cube” will represent the weapons on the robots
    - the “sphere” will represent any projectiles from the weapons, explosions, etc.
  + Also note: you can replace these models with whatever you’d like, so long as they are clearly different (even when they are really small on screen)

Your solution needs to be a 3D OpenGL solution.

I am **NOT** concerned with execution speed, so don’t worry about that. Well, within reason... like if it “locks up” for 30 seconds, then you really should look into how you are doing thing.

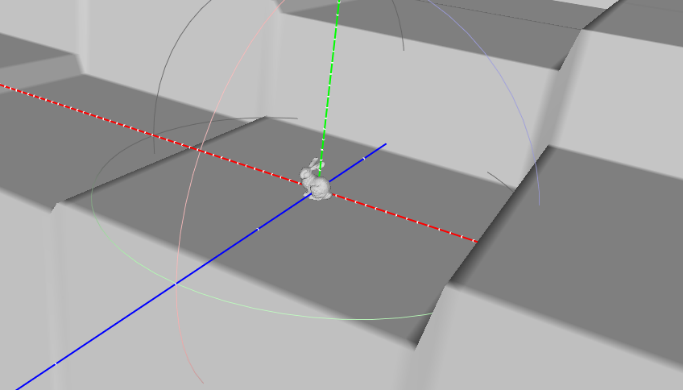
I’m also **not** concerned with how “pretty” it looks (lighting, etc.), just as long as you can see everything clearly. Like if you just have a single directional light in the scene, that’s totally fine.

1. (10 marks) Using MeshLab, generate an “arena” by doing the following:

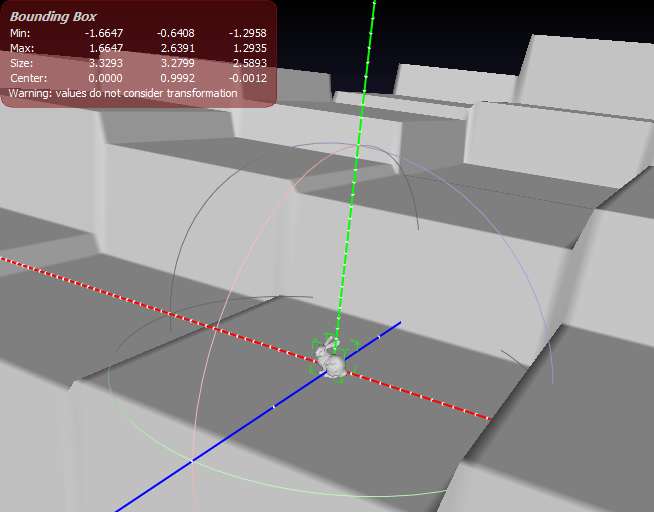
* Open MeshLab (without opening a model). This will open it with an empty “project”
* Choose “Filters”, then “Create New Mesh Layer”, then “Fractal Terrain”
* In the “Fractal Terrain” dialog box, choose “**Hybrid multifractal terrain**” (“Algorithm” dropbox.)
* Change the “Max Height” to **0.5**.
* ***Use your student number as the “Seed” value (default is 2.0)***
* Choose “Filters”, “Normals, Curvature, and Orientation”, then “Transform: Tranlate, Centre, set Origin”, then in the “Transformation” drop down list, pick “Centre on BBox”, then click “Apply”
* This will generate something like this 🡪   
    
  (Not the “boxy” stepped sort of look)

MeshLab assumes that “up” is “z”, so we need to adjust this. Turn on the “axis” drawing by choosing “Render”, “Show Axis” to make this clear (if you want).

* Choose “Filters”, “Normals, Curvature, and Orientation”, then “Transform: Rotate”.
* Type in “-90” in the “Rotation Angle”, leaving the “Rotation on:” set to “X axis”, and click “Apply”, which will get you this
* If you want the lighting to be correct, be sure to also choose “Filters”, “Normals, Curvature, and Orientation”, then “Re-compute Vertex Normals”
* Save this model with “vertex normals” selected and in ASCII format (“File”, “Export Mesh As…”, uncheck the “Binary encoding”)

1.  (20 points): There are 10 robots that will battle to the death, in the “arena”.   
     
   Place 10 robots (bunny models or equivalent) in random location in the “world”. They should be placed so that the base of the bunny models are “on the ground” (i.e. not floating in space). Choose an appropriate scale, but the bunny models should be quite small in comparison to the arena, like something like this 🡪   
     
   This model will be the “arena” where the robots fight.
2. (30 points) There is only one (1) type of robot, but a large variation of weapons and armour. Here’s a summary:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Weapon | Damage per shot | Reload rate | Notes: | Travel time | Damage Radius |
| LASER | 4% | 1 second | Line of sight | Instant | 1x |
| Bomb | 25% | 5 seconds | Ballistic | Until hits | 5x |
| Bullets | 2% | 0 seconds | Line of sight | Instant | 1x |

* The “reload rate” is how quickly the robot can take another shot, after the “projectile” has reached the target. So if the reload rate is “1 second”, then the robot must wait 1 second until shooting again.
* Note: a time of “0” means that the robot can shoot immediately
* “line of sight” means that the robot has to have a straight line shot to the target. If there is something in the way (the terrain), then the shot will not “hit”
* “ballistic” means that the projectile can hit anywhere, but shooting the projectile up in the air, and “lobbing” it towards the target (like the Red Birds in Angry Birds: <https://www.youtube.com/watch?v=q71RaKHn-iw>)
* The “Damage per Shot” is how much health a robot will lose if it is “hit”
* “Damage Radius” is how close the projectile has to be to the robot. Note this will depend on how large your model is, and is based on the largest extent of the bounding box that encloses your model (in Mesh lab, you can choose “Render”, “box corners” to see this; the largest “size” in the 3rd row of information, is the maximum extent, or length, of the bounding box).

.   
  
Each robot has 1 weapon, which is randomly chosen.   
  
  
  
To show this, place a cube above your model, of different colours:

* Red: LASER
* Yellow: Bomb
  + Green: Bullets

1. (100 marks) Have the robots start shooting at each other. Each robot has an “Update()” method that is called every “frame” of the simulation.

Each robot chooses the “closest” robot to it, that it can “see”.   
  
Note that the type of weapon will determine what the robot can hit. If the robot has a “line of sight” weapon (LASER or Bullet), then there will have to be a straight line between the robot’s weapon (approximately the “cube” above the robot) and the other robot.   
  
**“Line of sight” weapons:**   
  
You can determine if the robot can “see” another robot, but testing a sequence of points between the robot and the target robot, in this manner:

* + Take the target robot position, and subtract the current robot weapon position:   
      
    vec3 TargetToRobotVector = Target.xyz – robotWeapon.xyz;
  + Normalize this value to get the direction:  
      
    vec3 DirectionToTarget = glm::normalize(TargetToRobotVector);
  + Pick a very small value, like 0.1 (Assuming the terrain is 256x256), and multiply it by the DirectionToTarget. This will give you a “step size” that you will use to test (think “velocity”, but it’s not really velocity).   
      
    glm::vec3 MovementPerStep = DirectionToTarget \* 0.1f;
  + Pick a point at the robot’s weapon. This will be the starting point.   
    Run through a loop, adding this MovementPerStep to the position each time. At each step, check to see if the point is *below* the terrain at that point.  
      
    You do this by making a slight alteration to the “closest point to triangle” with the mesh:   
    1. When you do the “closest point to triangle” test, use a value of 0.0f for the height of BOTH the test point AND the triangle mesh.   
         
       This effectively makes the mesh “flat” and the test point is moving along a 2D surface.   
         
       The triangle that it returns is the one that the test point is closest to but *ignoring* the height above (or below) the surface.
    2. Return the *actual* height of that triangle.   
         
       So you are *comparing* by ignoring the height, but you *still want the actual height* of the triangle at that point. Note that the area is “flat” (a bunch of steps), so you can use any vertex of “the closest triangle” as the height.
    3. Compare that height with the test point.
    4. If, at any point in the loop, the point is *below* a triangle, you know that the robot *doesn’t* have “line of sight”.   
         
       If we get through the entire loop *without* being *below* any triangle, then the robot *does* have “line of sight” (can “see” the other robot).   
         
       You loop until the test point is *less than the distance* between the target robot and the robot’s weapon (the starting point).   
         
       If the robot can’t “see” its target, keep picking robots until it finds one it can “see” (has “line of sight”).

**No “line of sight”:**

If the robot has a “line of sight” weapon, but can’t see any other robots, then it will have to “teleport” to a new location. Pick a random location on in the arena, and test to see if there is a line of sight on at least one robot. If that location doesn’t work, keep randomly picking until you find one.   
  
Robots teleport instantly. If a robot teleports, place that robot at the new location.   
  
Note: it’s possible that a robot has targeted another robot, but then that robot has teleported. In this case, the “attacking” robot will have to adjust its target at the next update.

**“Ballistic” weapons:**

* + Robots with “ballistic” weapons can “see” any other robot, so picking a target is simply picking the closest robot.
  + The robot will have to pick an initial velocity of the projectile, something like 5.0f to start, which would mean 5.0 units *per second* of the simulation. Keep in mind that the Update() is the render update, which would happen many times (usually 60) a second, so you will have to scale that value per frame (you can either determine the *actual* frame time, based on the high resolution timer in glfw or an equivalent, or assume that the frame rate is fixed, and divide by 60.0 to get the “per frame” movement speed).
  + Pick a direction to the target. You do this like the “line of sight”, by taking the target robot position and subtract the attacking robot’s weapon position:   
      
    vec3 TargetToRobotVector = Target.xyz – robotWeapon.xyz;
  + Normalize this value to get the direction:  
      
    vec3 DirectionToTarget = glm::normalize(TargetToRobotVector);
  + The “ballistic” effect is done by setting a “velocity” value using the X & Z values to the DirectionToTarget \* PerFrameSpeed, but setting the Y value to a positive value of PerFrameSpeed.   
      
    In other words, the projectile will move towards the target in the X & Z values, but “fly up” with the initial Y value.   
      
    At each step, change the Y value by some “gravity” acceleration. Pick some “per frame” value that looks reasonable; it will likely be quite small (gravity is 9.81 m/s/s on Earth, but keep in mind that this isn’t Earth and a framerate of 60 Hz will reduce this “per frame” step by 60x, so 9.81/60 *at least*)
  + Like the “line of sight” calculation, you would loop through, starting at the location of the robot’s weapon, and moving the projectile (with the velocity and the acceleration) until it “hits the ground”. You can test for “hitting the ground” using the same technique you used for the “line of sight”: when the test point is “below” any triangle’s height, then it’s hit, and the ballistic trajectory is over.

**“Shooting”:**

Each robot finds a target and “shoots”. *Assuming the robot has a target,* show this in the following manner:

* For “line of sight” weapons, draw small spheres along the line of the test points, from the attacking robots weapon position to the target robot. You do this using the same technique you use for *testing* the line of sight, but you don’t need to bother testing for ground penetration (you can, though, if you want – it won’t make any difference).   
    
  This should look like a solid line from attacking robot weapon to the target robot.   
    
  Once a robot has shot, show this line for 1 second (or something like that), then wait (if the weapon has to reload), and the shoot again.
* For “ballistic” weapons, draw the arc of the projectile from the attacking robot’s weapon to the point it penetrates the ground, but drawing little spheres (the colour of the weapon) at each test point you used for testing to see if it penetrated the ground.   
    
  Assume that the ballistic weapons fly immediately. i.e. you don’t have to animate them “flying”.   
    
  It’s unlikely that the ballistic weapon will hit on the 1st try. When the ballistic weapon hits something, test to see how close the point was (comparing the distance from the attacking robot’s weapon and the target robot vs the distance from the attacking robot’s weapon and the impact point). If it “went long” (too far), reduce the Y speed (vertical speed) by 5% and try again. If it “fell short” (too close), then increase the Y speed (vertical speed) by 5% and try again.   
    
  If the ballistic weapon exploded *and damaged the target robot*, then keep the Y speed *the same* and shoot again.

**“Hitting” (and damage):**

* “line of sight” weapons always hit. If a robot is hit, it takes the appropriate amount of damage each “shot”.
* “ballistic” weapons explode when they hit the ground. Anything within the “blast radius” will be damaged by the appropriate amount. This can also include the robot that shot the weapon (self inflicted damage).

**“Dying”:**

* A robot has “died” if it has no health. They initially start with 100% health.
* Once a robot has “died” (<= 0% health), change the colour of the robot to light grey.
* Keep cycling until there is only 1 robot left.

1. (60 marks) Rockets…  
   * Now increase the simulation to 20 robots per game.
   * Add a 10% chance that a robot will be created with a “rocket” weapon, it takes 10 seconds between shots and delivers 30% damage in a “blast radius” of 7.5 x the size of the robots.
   * This weapon is a combination of the “line of sight” weapon and a ballistic weapon:  
     1. Robots will “target” another robot, but, at first, WON’T see if they are *actually* able to “see” the target robot. They will just pick the closest robot and shoot in a straight line.   
          
        You draw the path of this rocket using the “line of sight” method, but *stop* when the test point is *below* the ground (just like you did with the line of sight and ballistic weapons) OR if it hits another robot.   
          
        When the rocket “hits” (the ground or robot), anything within the “blast radius” *including the robot that shot the rocket* will take damage.   
          
        If the rocket hit the target robot, then the attacking robot should keep shooting at that target.   
          
        If the rocket hit the ground instead, then the attacking robot should pick the next closest target and try again.

**That’s it.**