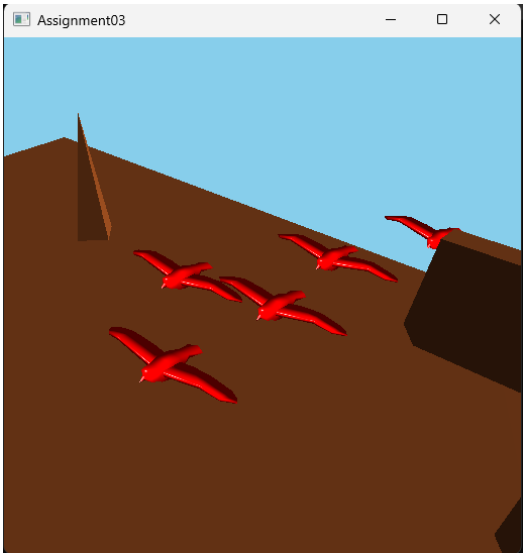
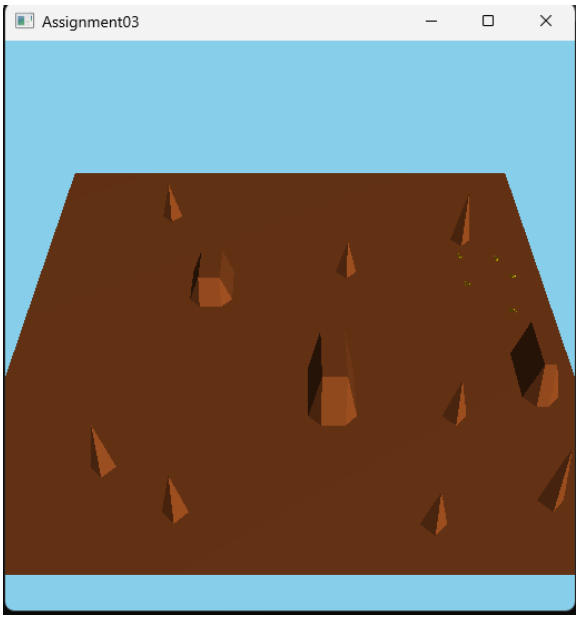
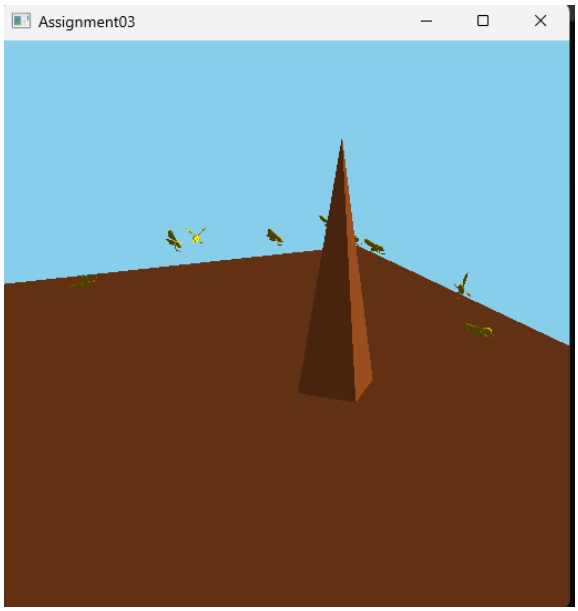
After careful consideration, I chose to work on the topic: **“Implement the Boids algorithm for flocking animation. Consider how this technique could be extended to other types of objects.”**

To begin, I revisited assignment 3 and replaced the monkeys with birds. I find it helpful to start a project and refine my ideas as I progress with the implementation.



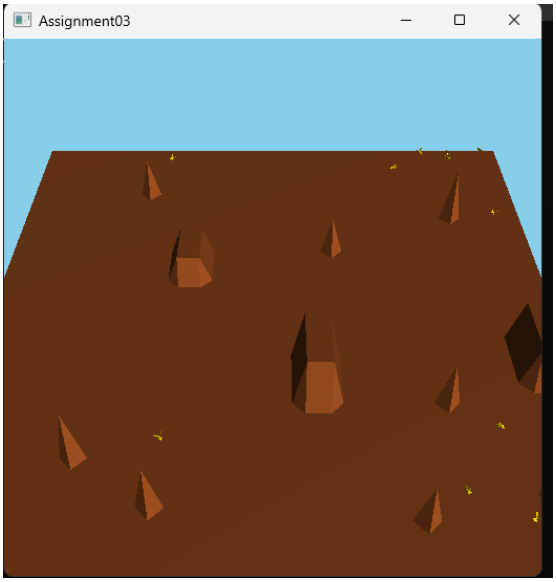
Shortly after, I decided that birds were not original enough and opted to make a change. I decided to create a bee swarm simulator, perhaps even a 3D one. This idea was inspired by some of the demos I found, when less structured they reminded me of the chaotic behavior of bee swarms.





I switched the model to bees, but since bees don't move smoothly, I added noise to their direction. Next, I had to adjust how the bees moved. My original implementation, based on monkeys, had them moving toward a goal. However, I wanted the bees to roam naturally. To achieve this, I replaced the goal-oriented, vector-based movement with physics-based movement, similar to what we used in Lab 9.

```
glm::vec3 noise = glm::linearRand(glm::vec3(-1.0f), glm::vec3(1.0f)) * noiseScale;
v += noise; // Add random noise

if (glm::length(v) > maxSpeed) {
    v = glm::normalize(v) * maxSpeed; // Limit speed
}

p += v * deltaTime; // Update position

d = glm::normalize(v); // Update direction
```

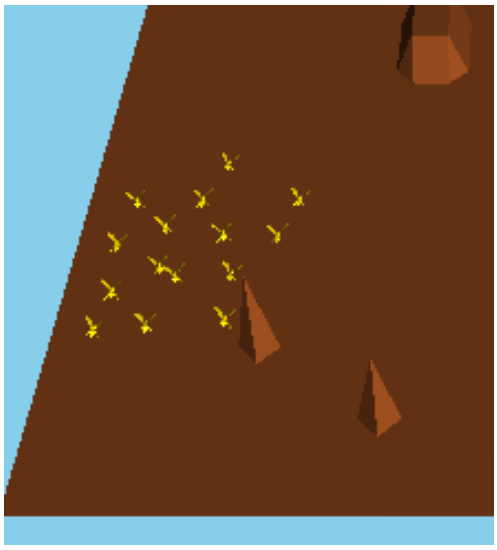


This left the bees wandering aimlessly, but the transition was successful, they still avoided obstacles, boundaries, and each other. With this foundation in place, it was time to implement some form of grouping behavior. I added a check where each bee evaluates the positions of all other bees within a 10-unit radius. It then calculates the average location of those nearby bees. A vector pointing toward this average point is generated and applied as an acceleration force, pulling the bees together.

```
private:
    // Calculate team average position and direction within a radius
    std::pair<glm::vec3, glm::vec3> teamAvg(std::vector<std::vector<glm::vec3>> posDir, float r) {
        glm::vec3 sumPos(0.0f);
        glm::vec3 sumDir(0.0f);
        float size = 0;
        for (size_t i = 0; i < posDir[0].size(); ++i) {
            if (glm::length(posDir[0][i] - p) < r) {
                sumPos += posDir[0][i];
                sumDir += posDir[1][i];
                size++;
            }
        }
        return std::make_pair(sumPos / size, sumDir / size); // Return average
    }
```

This is my final method for the avg of the bees' locations.

I later adjusted the radius to create multiple swarms of bees, making the simulation more dynamic. Now, each bee only looks for others within a 5-unit radius. This change resulted in the formation of several smaller swarms instead of one large group. To enhance the effect, I also averaged the heading vectors of the bees within the radius and applied the result to each bee, ensuring they moved in the same direction as their swarm.

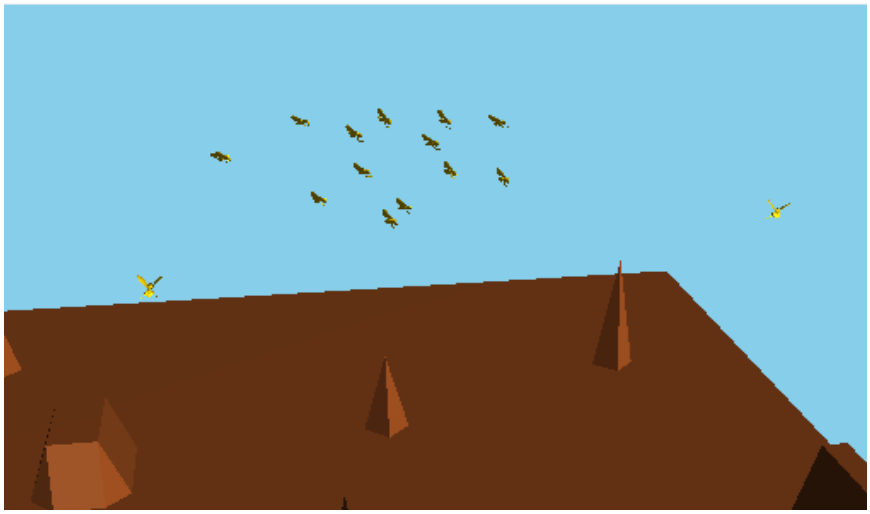


With the bees now swarming properly and all three rules of Boids applied, I wanted to push the simulation further by implementing the Boid algorithm in 3D. Previously, many of the y-values were routinely set to 0 to keep the movement confined to a 2D plane, as the original monkeys didn't fly. However, since bees do fly, this restriction no longer made sense.

The biggest challenge with this change was that my boundaries were not designed for 3D; they only existed in 2D. My wall boundaries were essentially just a line of invisible objects along the edges of the ground model, and extending this approach to 3D would have been computationally expensive. To address this, I redesigned the boundary system.

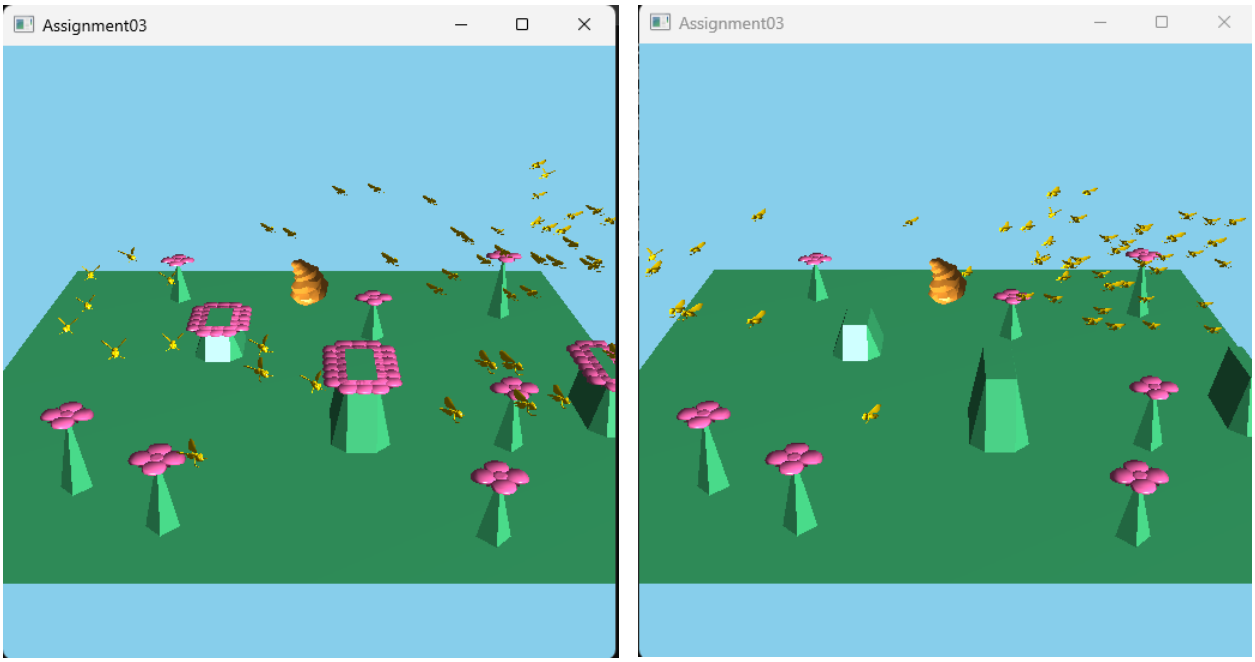
My solution involved identifying the maximum vertices of the model and storing them in a struct, which was then passed to the bees' movement checks. Once this system was in place, I updated all relevant vectors to include proper y-value calculations, enabling full 3D movement.

```
// Boundary repellent force
glm::vec3 boundaryForce(0.0f);

if (p.x < bounds.min.x + boundRad) {
    boundaryForce.x += contStr / (p.x - bounds.min.x);
}
if (p.x > bounds.max.x - boundRad) {
    boundaryForce.x -= contStr / (bounds.max.x - p.x);
}
if (p.y < bounds.min.y + boundRad) {
    boundaryForce.y += contStr / (p.y - bounds.min.y);
}
if (p.y > bounds.max.y - boundRad) {
    boundaryForce.y -= contStr / (bounds.max.y - p.y);
}
if (p.z < bounds.min.z + boundRad) {
    boundaryForce.z += contStr / (p.z - bounds.min.z);
}
if (p.z > bounds.max.z - boundRad) {
    boundaryForce.z -= contStr / (bounds.max.z - p.z);
}
if (p.x < bounds.min.x || p.x > bounds.max.x
    || p.y < bounds.min.y || p.y > bounds.max.y
    || p.x < bounds.min.z || p.z > bounds.max.z) {
    a += glm::normalize(glm::vec3(0.0f) - p) * maxSpeed * 1.25f;
}

// Combine forces into acceleration

a += boundaryForce;  // Add boundary repellent force
```



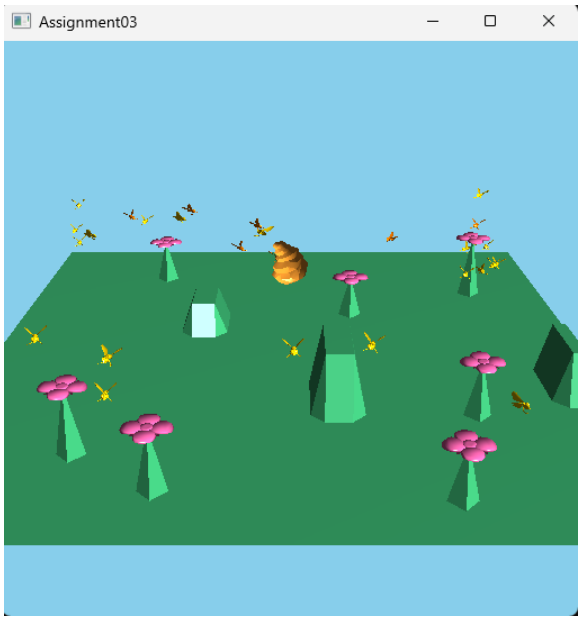
With the boundaries in place and the y-values properly functioning, the bees could now move in 3D. Next, I wanted to make the bees behave more like real bees by giving them tasks beyond aimless wandering. Using a testing feature I had previously implemented for the monkeys, I added models at all collision points. This caused flowers to appear at these

points, and I also introduced a beehive to the simulation. However, for the bees to interact with these objects, I needed to make them naturally attracted to them.



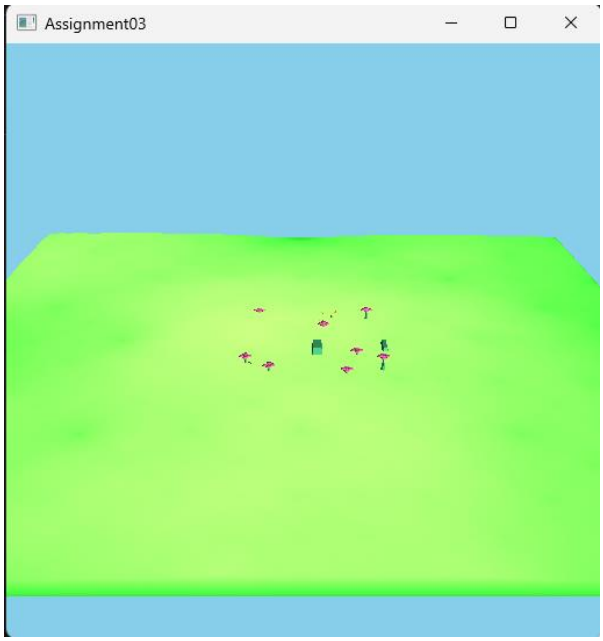
I implemented a feature to differentiate flowers from obstacles by first identifying isolated points. I resolved an issue by rounding the vertices to eliminate very close obstacle points. Then, I separated the isolated points from the obstacle points, leaving flowers positioned appropriately, as seen in the second photo above. The next step was to make the bees interact with the hive and the flowers.

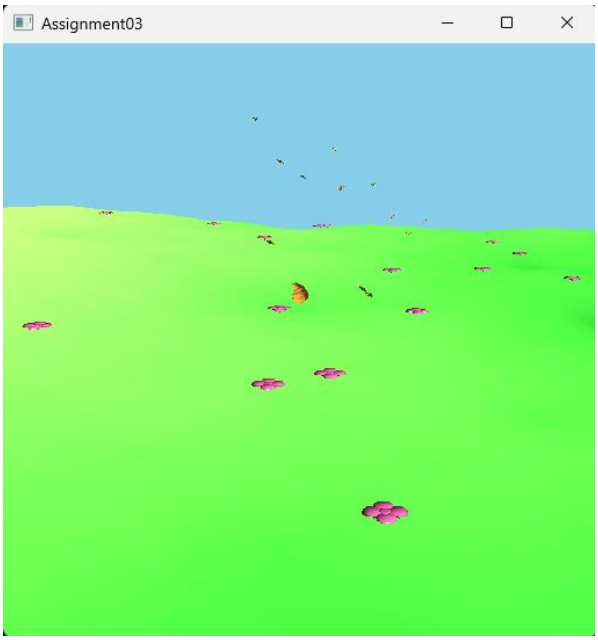
By adding flowers to a separate point list, I was able to adjust the bees' reactions upon detecting them. When a bee and a flower are within the same radius, a strong vector is added to the bee's acceleration, directing it toward the flower. This vector is only applied to acceleration, so the bees still experience noise and interact with other bees, making their movement appear more natural. Once the bees enter an even closer radius, they interact with the flowers. They also interact with the hive, returning to it after locating a flower.



When a bee touches a flower, it turns orange and begins its journey back to the hive. Upon reaching the hive, the bee turns yellow again. Each trip represents pollen collection, and the total amount of pollen collected is recorded. This number directly influences the hive's growth and the rate at which new bees are born.

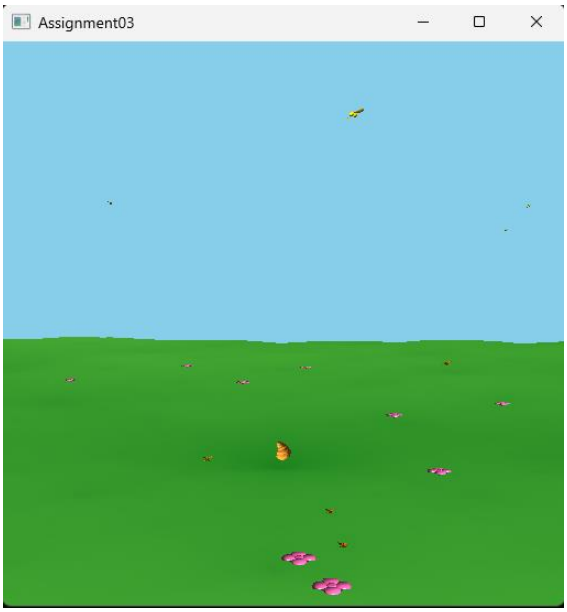
Finally, I wanted to scale up the simulation, so I replaced the flat ground with the fractal-generated terrain from Assignment 1. While it took some time to integrate, the effort was worthwhile. I did, however, remove mountains and water, as bees typically don't live or fly over these areas. If time permits, I may add these features later.





.

First, I needed to develop a new method to generate points and flowers, as they were previously based on the ground model, which was now bumpy and uneven like real terrain. This required updating the boundary calculations. I achieved this by using the height map to determine the height at rounded x and y coordinates. This approach allowed me to align objects with the terrain and prevent the bees from entering the ground.



Once the foundational work was completed, most other issues were resolved. For example, bees sometimes gained too much speed and overshot their targets. I addressed this by tweaking variables and adding caps to limit their acceleration and velocity. To maintain a relatively field-like appearance, I used a small preset height map.

After laying the groundwork, I spent significant time adjusting variables to find a balance between the Boids algorithm parameters, ensuring the bees' movement looked as realistic as possible. I'm very happy with the final result. Additionally, I implemented a feature that moves flowers every two seconds. This was achieved using a stack vector list, where the first flower is removed, and a new one is added in a different location.

While I'm pleased with the outcome, there are features I would have liked to include, such as trees to serve as obstacles and flower patches for a more realistic appearance. I also added a maximum limit for the number of bees and hive size to prevent the program from slowing down due to excessive computational load. It takes about 15 min to max out both.

Another thing I added was a class to control the camera changes. The controls for the simulation are as follows: use **W** and **S** to move forward and backward, **A** and **D** to move left and right, the arrow keys to adjust the camera's pitch and yaw, and press **ESC** to exit the simulation. Build instructions are in the read me. I've also included a demo video in the docs folder of the program running. The max bees and max hive are displayed below.