Jerzy Jaskuc **27.12.2020**

Computer Graphics Final Project Report

In this report I will go over some of the more important features of the code, explaining the ideas and design behind them, as well as some of the features that I may have not mentioned in the video.

I will also not go over things that I explained in the mid-term report already or extensively talked about in the video.

Crowd

Generally, in terms of crowd, number of the entities in the crowd is not specified (it's just a constant, which can be changed) and so we can work with any number of them. On the demo I worked with 7 of them, because I believe it's a reasonable enough number to showcase all of the features and for it to not take too long.

Boids

In order to reduce computational load of calculating boid behaviour for every crowd member in the scene, I modified the algorithm a bit, while still visually preserving the boid principles. In particular, I combined velocity matching and crowd centering into one action. Instead of calculating both separately, I have a crowd center which is outside of the crowd. Crowd will try to center around it and follow it and so, while it keeps the flock intact, it also sort of matches the general direction of the whole crowd. This reduces computation by approx. order of O(N) where n is number if crowd members, as we don't need to match to direction of neighbouring members.

In terms of priorities, the number one is to avoid the predator (player), then to avoid neighbours and environment objects, and lastly to join the crowd.

The priorities can have different strengths. The closer an object that it needs to avoid is, the stronger it tries to move away from it (i.e. its direction will be more and more influenced to point away from the object) and vice versa.

The change in direction is also scaled to give an impression of smooth turn, rather than sharp change of direction when the object enters the vicinity of the crowd member. Like for e.g. when returning to the crowd.

```
// going back to crowd center is lowest priority and should be smooth, 0.1 modifier
glm::vec3 crowd_direction = glm::normalize(crowd_center - crowdAnts[i].trs) * 0.1f;
ant_direction = glm::normalize(crowd_direction + ant_direction);
```

Materials

To showcase different materials, I created 6 of them, 1 for main ant and 5 for crowd ants. These are as follows :

- Ruby
- Obsidian

- Black plastic
- Black rubber
- Gold
- Pearl

The materials are assigned to crowd members in round-robin fashion and so if there are more than 5 crowd members, the assignment queue starts over again.

```
newAnt.mat_properties = crowdMats[fmod(i, crowdMats.size())];
```

Example of obsidian material:

```
// Obsidian
Material newMat = Material();
newMat.ambient = glm::vec3(0.05375f, 0.05f, 0.06625f);
newMat.diffuse = glm::vec3(0.18275f, 0.17f, 0.22525f);
newMat.specular = glm::vec3(0.332741f, 0.328634f, 0.346435f);
newMat.s_coefficient = 38.4f;
crowdMats.push_back(newMat);
```

I didn't add more materials as I believe it doesn't showcase anything additional, it's just an extra vector passed to an array. All of the material properties are then passed to the shader through uniform variables.

Animation

The small ants have the similar hierarchical animations to the main ant. However, since they are smaller, I also speeded up their leg movements a bit, to account for visual disproportions in speed when traversing the ground.

Rock Textures

The rock texture is taken from free online resources. However, I took a single rock and built borders and candle "stands", out of them myself, using Blender. The rocks have 2 textures rendered.

One is diffuse texture that gives rock its base colors (influenced by ambient, diffuse and specular light with small intensity, since rocks don't really have shiny highlights in real life).

The other one is the map of light normals. In order to give rocks a bit more detail, I used normal mapping in addition to regular lighting. Since some rocks were rotated, some of the normals also had to be transformed into tangent space, but the final result gave quite visible improvement. On the next page you can see the comparison, where the first image shows the shading without normal maping, and the second (bottom one) shows the same view but with normal mapping.

Shading

We use Phong shading and Phong illumination models in our scene. I believe it was unnecessary to use Blinn-Phong shading here, since we do not have that many elements and its not yet that computationally heavy, even with multiple light sources.

Light sources

There are 3 lights sources in total. First is directional light, which simulates moon light coming under the slight angle.

```
// Main light properties (directional)
uniform vec3 direction = vec3 (0.2, -1.0, 0.2); // Light position in world coords.
uniform vec3 ambient_color = vec3(0.03, 0.03, 0.03); // ambient light
uniform vec3 diffuse_color = vec3(0.7, 0.7, 0.7); // diffuse light
uniform vec3 specular_color = vec3(0.4, 0.4, 0.4); // specular light
```

The ambient light is kept low, to keep the “spookiness” of scene, by making it darker.

The other 2 light sources are point light sources representing candles. Unlike moon light, those have yellow colour, as well as they attenuate quite rapidly.

```
// Yellow color
candleLights[0].ambient_col = vec3(1.0, 1.0, 0.0);
candleLights[0].diffuse_col = vec3(1.0, 1.0, 0.0);
candleLights[0].specular_col = vec3(1.0, 1.0, 0.0);
// Attenuation for 100 block distance
candleLights[0].att_constant = 1.0;
candleLights[0].att_linear = 0.07;
candleLights[0].att_quadratic = 0.017;
```

All of the lights are added together for the final shading result.

Particles

In order to add more liveliness to the scene, I decided to add some particle effects. Depending on the type of the particle, they are spawned and updated every frame or only on certain events. However all of them follow same principle, where a particle has its lifetime and changes its color/velocity/transparency over its life period. When life of the particle runs out, it’s no longer rendered and updated, until it respawns. To not waste computations on searching for the particles to respawn, we save the index of last used particle so we can almost instantly find the next one to respawn, as It’s usually exactly next one or close to it.

Blood

When the big (player controlled) ant registers a collision with the crowd ant, the small ant disappears, leaving a splash of blood particles in its place, which drop on the ground as if they were pulled by “gravity” before disappearing. Blood particle spawn is also followed by a “splat” sound, hinting that the crowd ant got squished (sounds are imported using IrrKLang library). Blood particles in itself are just small red cubes, which become smaller and more transparent as their lifespan goes. Unlike fire particles, we spawn all 100 blood particles at the same time, but only at the moment of collision.

```
void RespawnBloodParticle(Particle& particle, glm::vec3 position)
{
    GLfloat rand_x = ((rand() % 10) - 5.0f) / 100.0f;
    GLfloat rand_z = ((rand() % 10) - 5.0f) / 100.0f;
    particle.position = position;
    particle.transparency = glm::vec4(0.6f, 0.0f, 0.0f, 1.0f);
    particle.life = 2.5f;
    particle.scale = 0.3f;
    particle.velocity = glm::vec3(rand_x, 0.0f, rand_z);
}
```

Example of particle update:

```
// update all blood particles
for (unsigned int i = 0; i < nr_blood_particles; ++i)
{
    Particle& p = blood_particles[i];
    p.life -= 0.1f; // reduce life
    if (p.life > 0.0f)
    {   // particle is alive, thus update
        p.velocity += glm::vec3(0.0f, -0.004f, 0.0f);
        p.position += p.velocity;
        p.transparency.a -= 0.03f;
        p.scale -= 0.01f;
    }
}
```

Fire

Both candles also emit particles that simulate fire (turning into smoke) in them.

Candle particles spawn at a rate of 2 per frame, with maximum of 37 particles to be alive at the same time. This allows for smooth continuous “fire” emission, while also keeping the separate particles visible, rather than just rendering them as a static stream.

```
void RespawnParticle(Particle& particle, glm::vec3 position)
{
    GLfloat rand_x = ((rand() % 10) - 5.0f) / 170.0f;
    GLfloat rand_z = ((rand() % 10) - 5.0f) / 170.0f;
    particle.position = position;
    particle.transparency = glm::vec4(1.0f, 1.0f, 0.7f, 1.0f);
    particle.life = 2.0f;
    particle.scale = 0.2f;
    particle.velocity = glm::vec3(rand_x, 0.1f, rand_z);
}
```

Collisions

Borders

In order for all of our ants to not go outside of the map, there are invisible walls implemented on each of the 4 directions, more or less at the point where the natural rock borders start. Crowd ants will try to move away from the border, exactly same way as they do with neighboring ants or player ant.

In addition to that, both crowd and player ants cannot pass the border. If they are on the border, they do not advance the respective coordinate, but can still move in other directions.

```
// Border safety translation
if (crowdAnts[i].trs.x > x_border) {
    crowdAnts[i].trs.x = x_border - 0.1f;
}
else if (crowdAnts[i].trs.x < -x_border) {
    crowdAnts[i].trs.x = -x_border + 0.1f;
}
if (crowdAnts[i].trs.z > z_border) {
    crowdAnts[i].trs.z = z_border - 0.1f;
}
else if (crowdAnts[i].trs.z < -z_border) {
    crowdAnts[i].trs.z = -z_border + 0.1f;
}
```

Middle Rocks

Bordering off middle rocks proved to be much trickier than initially thought. Since the rocks are imported as one object, they do not really have separate coordinates and neither do they have a nice square shape, that could be used as a border.

Therefore, in terms of crowd ants, there are no hard borders. Instead there are multiple points taken from along the borders and these points act as objects that should be avoided by the ants. Like with any object avoiding, the closer ant is, the stronger is repel. With the rocks, if ant gets too close to one of the points, the scale at which it affects ant's direction is on par with predator avoidance, so that ant will go to the side from that rock at most, if not completely avoiding it.

```
// middle rocks
std::vector<glm::vec3> collision_points = {
    // rock 1
    glm::vec3(6.8f, 0.0f, -15.9f),
    glm::vec3(17.75f, 0.0f, -20.0f),
    glm::vec3(18.0f, 0.0f, -12.5f),
    glm::vec3(11.24f, 0.0f, -21.04f),
    // rock 2
    glm::vec3(-6.35f, 0.0f, 5.5f),
    glm::vec3(-19.2f, 0.0f, 7.5f),
    glm::vec3(-20.0f, 0.0f, 17.35f),
    glm::vec3(-5.2f, 0.0f, 11.03f),
};
```

For player ant I split the rock into three squares, which together more or less resemble the rocks shape. Together they act exactly as the border, preventing from moving into them.

Player to Crowd Ant

On the contrary, detecting collision between crowd and player ant is quite straightforward. Since ants silhouette resembles a rectangle we can just calculate offsets to the edges of the ants and if those hitbox rectangles intersect with each other, we score a collision, which sets off some of the events.

To even further simplify calculations, I utilize *glm::distance* function, since in practice player will always face towards crowd ant if it wants to catch it.

Text Rendering

In order to gamify our scene a bit, I implemented text rendering and timer. With that, the main objective is to catch all of the ants as fast as possible. The text is rendered using the help of FreeType library.

Time measurement

In order to properly measure time, I use *glutGet(GLUT_ELAPSED_TIME)* on every frame, once the game starts. Since this is executed on every frame, we subtract the time from previous frame to get the delta of time. Then I convert it into seconds and add to the main counter. Once the game is finished we no longer add to the counter and just display the final time.

```
int timeSinceStart = glutGet(GLUT_ELAPSED_TIME);
int deltaTime = timeSinceStart - oldTime;
oldTime = timeSinceStart;
time_spent += (GLfloat)deltaTime / 1000.0f;
std::string t = std::to_string(time_spent);
render_text("Your time: "+t, time_spent_coords, 0.5f);
```

High Score Saving

If at the end of the game, new time is smaller than the best time, I save it into the file, overwriting the previous one.

```
// save new best score
void save_score() {
    std::ofstream ofs("../../Lab1/high_score.txt", std::ofstream::trunc);
    ofs << best_time;
    ofs.close();
    cout << "High score saved" << endl;
}
```

The file is loaded every time the game is started, so that we don’t lose our best score, even when restarting the game.

```
// load the best time from the file
void load_best_score() {
    std::ifstream file("../../Lab1/high_score.txt");
    std::string str;
    std::getline(file, str);
    std::istringstream in(str);
    float score;
    in >> score;
    best_time = score;
    cout << "High score loaded" << endl;
}
```

Start and End of Game

I restrict the ant and camera movements until the user presses ‘ENTER’ to start the game, as well as when they catch all of the ants. This way it feels like the game has a menu and does not randomly start when everything is loaded. I believe it feels nicer in a way that it actually interacts with the user and not just displaying things on the screen.

Video Demo

Link to demo: https://youtu.be/1-WZxfj3iWk