Ant Attack

Overview:

Ant Attack is a top-down strategy/tower defense game where the player's goal is to destroy the attackers (ants), before they fully destroy the player's tower (sandwich). The game is set on a picnic blanket in a park, surrounded by tall grass. In the middle of said blanket is a plate with a sandwich, which the ants are trying to get to. The player must use a magnifying glass to fend off waves of ant attackers, before the sandwich gets fully eaten.

Features:

The insects spawn in waves. Each wave is represented by a level number and each level, the number of attackers grows. The first level/wave serves as an introduction to the mechanics of the game, and spawns 10 ants, the second spawns 20, all the way until level 10, where the player must fend off 100 ants. It is impossible to lose the game on level 1, as the base sandwich health is 100, and each ant only deals 5 damage.

Dealing damage:

Speaking of losing, to prevent this outcome, the player must direct a light-spot, representing the magnifying glass beam, around the blanket, and line it up with the insects in order to damage them. This is done by using the mouse cursor, which has been turned off inside the game-window, and its position inside the window mapped to the blanket size, meaning the player cannot leave the blanket with the light-spot. The magnifying glass object rotates and tilts, following the current position of the light-spot. Currently, the light-spot is always being projected onto the game-plane straight from the top (-y axis), instead of having a fixed position behind the magnifier and projecting the beam at an angle. This was done, because the latter implementation heavily distorted the beam, the closer it got to the edges of the blanket, and the light-spot thereby lost its characteristic shape. It was also much harder to aim the beam.

Stun:

To aid the player in his struggle, and to avoid being completely overwhelmed in the later levels, we have added a stun ability. Every 5 seconds, the player can press space, to effectively stun every insect attacker currently in the game for a specific amount of time. At the start of the game, this stun lasts 1.5 seconds, but can be (theoretically) upgraded indefinitely. As a visual effect supporting this ability, the light-spot radius quickly enlarges, engulfing almost the whole blanket in light, before quickly shrinking back down, supposedly signifying a sort of flash. (This ability also has a cooldown icon in the HUD, informing the user about its availability).

HUD:

In the desire to keep the HUD as minimalistic as possible, we decided not to have the sandwich health displayed in the form of a health bar. Instead, multiple models were used, with varying stages of missing chunks, to signify the health dropping. Health bars on every individual ant would lead to too much visual clutter, so we decided against that too, since it normally does not take long enough to kill one, to even notice a potential health bar.

Developed by Wolfgang Pillwax & Patrik Szabo

Health recovery / Breadcrumbs:

When an ant manages to "take a bite" out of the sandwich, it takes a little crumb of it with him on his journey back into tall grass, represented by a small yellow low-poly sphere hovering above the ant. Ants carrying breadcrumbs like this can be eliminated, to recover some of the sandwich health (2.5HP). This is important because sandwich health does not normally regenerate after each wave. This brings us to the upgrade phase.

Upgrades:

After each wave is over, either by killing all the ants or the ants escaping with the food, an upgrade screen appears. It gives the players 3 upgrade options:

1. Restore the sandwich health to full (100HP) -> Number 1 key on the keyboard
2. Increase the stun time by 0.25 seconds -> Number 2 key on the keyboard
3. Upgrade sunbeam damage multiplier by 0.5 -> Number 3 key on the keyboard
4. Hard mode (Do not upgrade) -> ESC key (only when in upgrade mode)

After some testing, we decided to make the number of "levels" of each upgrade theoretically uncapped. I say theoretically because the game ends after level 10. This was not always the case, and we experimented with the game being score-driven, meaning there would not be levels, and players would compare their skill based on how many ants they have managed to kill. Eventually, we concluded that this traditional win/lose level system fits the concept better, but the upgrade system stayed to allow the players as much freedom as possible with how they approach each level. Will they choose to fully upgrade their damage, to disintegrate the ants in the blink of an eye? Opt for a longer stun to carefully think about their next move? Will they deem it worth it to heal their sandwich back to full, or would an upgrade be better suited for the situation? It's not much, but more choice is never a bad thing. And if the player is looking for a real challenge, they can always forfeit their option to upgrade by hitting the ESC key. (Warning: this only works when in upgrade mode)

Developer features:

In addition to the standard developer/debug options provided to us with the framework (Wireframe mode F1 & Backface-culling setting F2), we implemented a couple of our own. Since the camera in-game has to be static, we locked the free-cam mode behind the F3 key, allowing the user to move around the "game-world" freely, with the game paused. Once disabled, the camera automatically snaps back to the position it needs for the game to continue. The F4 key enables and disables all the shadows in the scene, and F5 does the same with the in-game HUD, both of which will be further discussed.

Implementation:

The basic functionality was achieved using some good old key callbacks, matrix transformations and math. For the ants, a random side is picked, and based on the blanket dimensions, an ant is spawned there, with a distance multiplier, based on the wave size. Every frame, the ant moves along its position vector closer to the middle, based on a predefined speed value multiplied with the frameTime to account for different frame-rates on different machines (frame rate independence, this was also done for the damage dealt to ants, and the grass swaying in the "wind"). Once the ant passes a certain point on its journey to the sandwich, its y position changes to account for the plate elevation and finally after it

has reached the sandwich, it gets flipped by 180 degrees and runs back. Ants, among other things, are saved in a Vector, and once all ants are either dead or escaped, the vector becomes empty, signifying the end of the round. The upgrade mode gets triggered, and after an upgrade has been chosen, a new round begins. The attacking ants were first tested with just a set of small cubes, but soon enough, we needed proper models in the game.

Object loading with ASSIMP:

Following the learnopengl.com tutorial on model loading, we compiled an ASSIMP library to use in our project. The tricky part here was implementing a getMaterial function, that would extract the object material data from the mtl file generated by blender, for objects that did not have a texture. This material would then be passed on as a Uniform to the fragment shader and used if needed.

HUD:

The HUD is pretty basic. We created 2D-Quads and added textures on to them. To get a 2D-Quad we just use 4 vertices with no depth-coordinate which we then never transform into world-space. This way, the quad is seemingly stuck to the “lens” of our camera. 2D-Quads are assigned the desired texture and are only scaled and translated to the desired location. The HUD can also be toggled with the F5-key.

Instanced rendering:

To render tens of thousands of grass blades and hundreds of ants, we couldn’t realistically render each one individually with a separate draw call. Instanced rendering was the solution to this problem. Only one “object” gets actually sent to the GPU, along with the number of instances we want, and based on the instance number that gets automatically assigned to each instance, we can transform them using a matrix Vector of different positions, passed to the vertex shader in a uniform.

Vertex animation:

Apropos grass, the blades shouldn’t just stand in the background, motionless. A quite convincing swaying-in-the-wind effect was achieved using a combination of instance transformations(slight rotation along the x axis to simulate the tilt) and vertex animation, where, based on current time measured and using a sine function, the blades’ vertical position changed in the vertex shader.

Shadows from a point-light using an omnidirectional shadow map:

Again, thanks to a couple youtube tutorials and learnopengl.com, we were able to implement shadows cast by the point-light representing the sun. They range from very faint, like on the small ants, to noticeable ones like on the sandwich and grass. This effect is best observed by toggling the shadows on/off with F4. Because we are dealing with a point-light that theoretically shines in all directions, the shadows/depth values were captured in a cube map. In the render loop, all relevant objects are first rendered into the depth map frame buffer, then in the normal render call, the fragments are compared against this depth map, and if the fragment’s depth is higher than that in the depth map, we know that it is in the shadow. Later, the so-called Percentage-Closer Filtering (PCF) was used to get rid of any artifacts and smooth out the shadows. A geometry shader was also used, which allowed us to build the depth map in just one render loop. This also made us write our own shader class, to deal with additional shaders like the geometry shader, or later the compute shader, since the framework Shader class doesn’t support those.

Simulating reflections with a cube map:

Generally to get reflection you only have to reflect the view-direction-vector around the normal-vector(we used the GLSL reflect-function) of a given vertex/fragment and then use this reflected vector to sample a given cubemap which usually would just be the skybox. The problem is that this would only result in the “reflection” of the skybox. To get a reflection of the whole scene we had to render a simple version (no shadows,..) of the scene and then create a cubemap at the position of the reflective model. To achieve this we first placed a camera-object at the position of the model, that is supposed to have a reflection. We then used a framebuffer to write the current frame onto an empty cubemap-texture. Then we rotated the camera 90 degrees and repeated the process. At the end of this process we received a cube map which had the skybox as well as a simple version of our scene on it, which we then used to simulate reflection on a knife and a thermos bottle in our scene. This can be best viewed with the free-cam (F3).

Smoke particle system using a compute shader:

Even though we could not finish the particle system it is still currently implemented. The basic idea is to create particles using the compute shader and then drawing these particles using a normal shader-program consisting of a vertex-, fragment- and geometry-shader. The vertex- and fragment-shader have usual functionalities and the geometry-shader is used to make a quad out of every particle. A texture is then projected onto the quads. To see the current results you would have to remove the comments from the commented functions startComputeShader(dt) and drawParticles(camera, smokeTexture) in the render-loop and then disable culling when the game is running. Although the current smoke textures are just placeholders.

Cel-shading:

Cel shading has been partially implemented as a test to see how the style would fit into the overall concept, by first transforming the rgb value of a fragment into hsv, taking the brightness value v, and defining “brightness steps” to which each brightness value would get converted, thereby transforming the otherwise linear brightness fall-off to statically defined values. The results were not great, so we decided against using it. The commented-out skeleton of cel-shading remains.

Sources:

- https://learnopengl.com/Model-Loading/Model
- https://www.youtube.com/watch?v=W_Ey_YPUjMk
- https://www.youtube.com/watch?v=ZbnEMM7vwmU
- https://learnopengl.com/Advanced-OpenGL/Instancing
- https://www.youtube.com/watch?v=6PkjU9LaDTQ
- https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/Animation_SS18.pdf
- https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows
- http://ogldev.atspace.co.uk/www/tutorial43/tutorial43.html
- https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/GPU_Particles_SS18.pdf
- OpenGL 4: Shading language cookbook 2nd edition
- ECG & CG Kurs der TU-Wien

Developed by Wolfgang Pillwax & Patrik Szabo