Praktikum: Echtzeit Computergrafik

Assignment 10 – *Shooting Enemies*

In this assignment, we will finally blow stuff up – our cannons should fire simple projectiles based on the already implemented sprite rendering, we will check for collisions with enemies, track their life points and make things go boom. Well, actually, next week we'll make things go boom.

Guns, Configurations and Sprites (1P)

*Your game should support two different, configurable guns: The **gatling** and the **plasma** gun. The configuration for each gun must contain: projectile speed, fire rate (respectively the cooldown time between two fired shots), damage inflicted on hit, and the amount of gravity influence on the projectiles (i.e. the particle mass).*

Furthermore, each gun needs to contain rendering information for its projectiles: a sprite texture, a sprite radius and a spawning position for new projectiles in view space.

- To allow sharing sprite textures between guns, a variable list of texture filenames should be given in the game configuration file. Add the corresponding functionality to your initialization – the retrieved filenames should replace the hard-coded ones from the last assignment, so pass those to your sprite initialization to create all needed resources.
- Create a `struct` with all the above gun properties and two instances for your guns. For sprite texture identification, you can simply use an index in the sprite list – so make sure your sprite textures are added in the correct order!
- Read all properties of each gun from the configuration file during your initialization.

Checkpoint: Make sure your code compiles and all gun parameters and projectiles are initialized correctly.

Projectile Spawning, Update and Death (3P)

Now it's time to fire your guns! There will be no specific checkpoints below, but it's very wise to check that your code still compiles after each step.

- Along with the static configuration parameters, each gun needs to keep track of its current dynamic state, i.e. if it is currently firing (spawning projectiles) and the cooldown time left until the next projectile is released. Add this to your code; each gun should be able to fire separately controlled by the player, so you have to assign specific keys to each of them (hint: use `OnKeyboard()` and store the value of `bKeyDown` in your `Gun`-struct). Write down your keybindings in your readme.txt!
- In each frame: check for each gun if a new projectile needs to be spawned based on the gun's cooldown timer (it's totally sufficient to update this once per frame). If so, add a new projectile to a global container (`std::vector` or `std::list`) at the initial position given by the gun configuration. Remember to consider the current view direction and position of your camera when you set the projectile's velocity and spawning position – use the inverse view matrix to transform the local spawning position to world space; you get this directly from `camera.GetWorldMatrix()`.

- Update all of your projectiles each frame using the projectile speed and gravity influence from the gun configuration. You already know how to do this – see the according lecture slides on projectile motion (Euler integration is sufficient here)!
- Make sure your projectiles do not live indefinitely – you could achieve this using a fixed lifetime or a maximum distance from the spawning position.

Projectile Rendering (3P)

- On rendering, collect all projectiles and pass them to your sprite renderer. Remember that you created sprite parameters (texture & size) for each of your guns!
- To assure correct blending, all sprites need to be sorted from back to front before rendering. To compare the view-space depth of two projectiles, simply calculate for each of them the dot product between their positions and the "ahead"-Vector of the camera (`g_Camera.GetWorldAhead()`) – the larger this value, the further away is the projectile from your camera. Sort your sprites accordingly (there is a `std::sort` method that you can use, check the documentation and the slides).

Checkpoint: You should now be able to fire visible projectiles from your guns. Make sure your projectile update and all parameters are correct – play around with the parameter settings until you're satisfied with the deadliness of your guns!

Collisions and Enemy Death (3P)

To shoot down enemies from the sky, a simple collision model based on bounding spheres will be implemented next.

- For each EnemyType, a `size` configuration parameter exists in your `game.cfg`. Make sure this parameter is read during your initialization – this is the size of the bounding sphere you'll be using for collision detection.
- In each frame, check all projectiles for an intersection with all enemy ships. To do this, simply test the bounding sphere of a ship against the bounding sphere (e.g. projectile radius) of a projectile; if the projectile is (partially) inside the ship's bounding sphere, a hit is detected. It's sufficient to check the current position, but make sure your projectiles aren't moving too fast – you might miss intersections! See the lecture slides on collision detection for the corresponding formulas.
- Whenever a ship is hit, remove the respective projectile from the scene and apply the damage given by the configuration of the projectile's gun to the ship.
- Finally, whenever an enemy's hitpoints are reduced to 0 or less, remove the enemy from the scene. Die, alien scum!

Final Checkpoint: You should now be able to fire projectiles from both of your guns and hit enemy ships. Check if enemies are correctly hit & destroyed, and make sure your projectiles are also correctly deleted upon hit. Again, tweak all your parameters until the game "feels" right. After you are done, check your program again for memory leaks or unreleased resources as always.