
Computergraphics 186.831
Submission 2
CORE
Dominique Grieshofer 1054878
Christoph Derwart 1126977

Description

COR3 is a first-person arena shooter about defending mankind's last reactor core from hordes of attacking aliens. How long can you delay the inevitable destruction of COR3?

Story

In the distant future an unknown species is attacking earth. Several reactor cores were built far below the surface to power the planet's defense systems. After a devastating attack all but the third core, COR3, have been destroyed. Now you as the guardian of COR3 are mankind's last stand.

Features

- Single-player
- Two completely different weapons
- Two strategically different enemy types
- Fluid first-person movement
- Tense atmosphere

Controls

Key	Action
Move mouse	Look around
Left mouse button	Projectile stream
Right mouse button	Projectile burst
W/A/S/D	Movement
SPACE	Jump/Double-jump
T	Slow-motion: On/off
ESC	Main Menu
F2	Frame-time: On/off
F3	Wire-frame: On/off
F4	Texture sampling quality: Nearest neighbor/bilinear
F5	Mip mapping quality: Nearest neighbor/linear/off
F6	Debug camera: On/off
F7	Performance-test: Spawn all enemies
F8	View Frustum Culling: On/off
F9	Transparency: On/off
F10	Effects: On/off

Gameplay

The player can smoothly move around a small arena and shoot enemies using a continuous projectile stream or fire many at the same time using a projectile burst. Advanced movement includes a double-jump which can be used once before landing and a projectile-jump by shooting a projectile burst into the ground just after jumping off.

There are also two strategically different enemy types: Swarmers which move towards and damage the player on collision and Spawners which spawn more Swarmers.

Implementation

Freely Movable Camera

The camera is fixed to the player and is controlled using mouse movement (first-person perspective). The player can freely move in 3D space within the arena by jumping, double-jumping, projectile-jumping, strafing, running forward and backwards.

Moving Objects

All objects in the game are dynamic and move and/or rotate:

- Swarmer enemies move independently in 3D space and home towards the player.
- Projectiles move forward based on own rotation in 3D space and are reflected and therefore changing their rotation when hitting the arena edge or floor.
- Spawner enemies, the sun, COR3, COR3 circler, and the arena rings rotate around their z-axis.
- COR3 and its circler moves in a circular fashion around the whole arena.
- The spring-based floor moves vertically on projectile hit, player jump or landing.

Animated Objects

Although detrimental to the core design of the game (simplicity) there is a hierarchical animated object that rotates around the moving and rotating COR3 which was done to fulfill the submission requirement. This object will only be spawned and can therefore only be examined by using the debug camera (F6) which can then be seen just outside the playable area.

Texture Mapping

All enemies, arena rings and the arena floor are textured. Textures are loaded from .dds files using GLI and mapped with uv-mapping. Meshes, uv-mapping and normals are loaded from Wavefront .obj files using Tiny obj loader.

Lighting and Materials

All objects have materials. Projectiles and COR3 are purely emissive models. All other models are rendered with a BlinnPhong shader and provided with normal vectors. One directional light (sun) lights the scene which has independent ambient, diffuse and specular colors and changes color with player health.

Instanced Rendering

To improve performance with a huge number of enemies we are drawing everything apart from single objects like the UI using OpenGL instancing which reduces the amount of needed draw calls immensely.

Multi-Threading

Since there can be thousands of enemies we have parallelized the game engine update loop to be able to fully utilize all cores in modern computers. To be able to do this safely the game objects are not allowed to modify any states which are non-private which means that calculated values like transform changes are cached in a private variables and only applied in the non-parallel game engine update loop.

Additionally a thread pool is used which spawns as many threads as there are cores (minus one due to the main thread) on game start and then uses a queue to feed it more work when needed.

Collision Detection

Detecting collision between the player, enemies and projectiles is done discreetly each frame using bounding spheres. Additionally both the player and swarmer enemies also check for floor collision using simple z-axis comparison and for arena edge using 2D length comparison for x- and y-axis since the playable area is an infinitely high cylinder centered in the origin of the world.

Effects

- Spatial Hashing (1P)
- Shadow Maps (PCF) (1.5P)
- HDR (1P)
- Bloom (1P)

Spatial Hashing

Swarmer enemies move towards the player and to prevent clumping they check on each frame to avoid other nearby enemies. Since there can be thousands of enemies this is done using spatial hashing to improve performance by only having to check bounding spheres of enemies in same spatial grid. The same technique is used for collision detection between projectiles and enemies as well as the player and enemies.

The implementation is based on this article and optimized and simplified further for our use-case. We use a fixed grid size using pre-cached vectors and are also ignoring object size when storing and updating the position in the grid to not have to deal with duplicates in multiple grid cells when checking them. Downside is that the search radius must then be at least the same size as biggest object in grid to not miss any which is why different enemies have a similar size.

Shadow Maps

Shadow mapping and PCF is implemented as proposed in the texture mapping slides of this course. With the exception that the depth map for shadow mapping is rendered from a virtual light position since the shadow caster is a directional light. Shadow acne is prevented by enabling OpenGL polygon offset and Peter Panning by front-face culling based on this article. Because all shadow casters in the scene are solid objects we can use this solution.

HDR

The scene is rendered in HDR with tone-mapping and gamma correction. We use a 16bit RGBA floating point graphic buffer for all main and post processing render passes to get a reasonable dynamic range without taking up to much graphics memory. Tone-mapping is done with a modified version of the Amnesia/Uncharted 2 filmic tone-mapping operator described here and here. The last fragment shader stage also performs a simple gamma correction. Additionally there is a Chromatic Aberration effect which simulates non-perfect images of a real world camera lens.

Bloom

We use a basic HDR Bloom with a 5x5 pixel gaussian blur. The blur is separated in horizontal and vertical runs to increase performance. The implementation is based on this article.

View Frustum Culling

We use the geometric view frustum culling algorithm proposed in this article which improves performance only slightly in our case since we already use instancing heavily to render a large amount of objects which is already extremely fast on modern GPUs.

Transparency

COR3 is being rendered transparent which means that you can see other objects through it. This is done by rendering it after all other objects have been drawn and then rendering both the back- and front-faces of COR3 in that order. This object will only be spawned and can therefore only be examined by using the debug camera (F6) which can then be seen just outside the playable area.

All Used Libraries

OpenGL Mathematics (GLM) for math functions (https://github.com/g-truc/glm)
OpenGL Image (GLI) to load .dds files (https://github.com/g-truc/gli)
Tiny obj loader to load .obj files (https://syoyo.github.io/tinyobjloader)
GLFW for window and input handling (https://github.com/glfw/glfw)
GLEW to expose OpenGL functionality (http://glew.sourceforge.net)
ImGui for UI functionality (https://github.com/ocornut/imgui)
FMOD as the audio engine (http://www.fmod.org)

Used Tools

Visual Studio 2013 for programming (https://www.visualstudio.com)
Blender for modelling (https://www.blender.org)