

Documentation

Group/Game Name: Last tank

Github Link: <https://github.com/shiermann/ptvc26-lasttank>

Students: Alexander Cech 12125439, Stefan Hiermann 00671289

Brief description of implementation:

Our project uses an Entity-Component-System architecture. Which means, each object in the game is an entity with an integer ID, has some components that store data and dedicated systems that processes matching entities for each frame.

Physics is handled by an additional library called Bullet, where it walks through the simulation and writes the new positions into the ECS to report collisions so that the tank does not drive into things. The terrain of the game is generated from a heightmap. It is used as a tessellated patch grid for rendering as well as a collision shape for the physics engine. To render the game, we use a deferred pipeline. First in the pipeline is the shadow map pass, which renders the scene depth from the directional light's POV. Then the geometry pass fills the G-Buffer with world-space positions, normals, and albedo, ambient occlusion, metallic and roughness maps. Third, the SSAO pass determines how much ambient light each spot should get based on its surroundings, followed by a blur pass to smooth the result. Then the lighting pass uses a full-screen quad to read the G-Buffer, shadow map, and blurred SSAO to produce the actual image on screen.

Finally, particles, HUD, and text are rendered as forward passes on top.

Assimp is used to load the 3D models and DevIL is used for all texture and image loading like the hearts in the health bar or loading the heightmap for the terrain.

Particles will spawn when a weapon is fired, always face the camera and fade away as they die.

The player's tank can be controlled via WASD. To aim the enemy, use the mouse cursor and shoot with the left mouse button. The camera's rotation can be changed with the arrow keys and the zoom factor with the mouse scroll wheel. Two basic enemy type AIs that hunt the player are implemented and a health system where tanks only take damage from enemy projectiles. A HUD shows a health bar and a menu state machine deals with changing between the main menu, the actual game and the win/lose screens.

Additional libraries:

- Assimp (<https://github.com/assimp/assimp>)
for 3D model/object loading
- FreeType (<https://freetype.org>)
for font rendering
- Bullet Physics (<https://github.com/bulletphysics/bullet3>)
for rigid body physics & collision detection
- DevIL (<https://openil.sourceforge.net>)
for image loading

All packages are managed via vcpkg

Effects

Gameplay:

Mandatory:

- 3D Geometry (6 points): Player and enemy tanks, Projectiles, Terrain
- Playable (3 points)
- Min 60 FPS and Framerate Independence (3 points)
- Win/Lose Condition (3 points):
 - Win: Player destroys all the enemy tanks.
 - Lose: Player is hit too many times and destroyed.
- Intuitive controls (2 points)
 - Cursor position: Turret rotation.
 - WASD: Tank movement independent of the rotation.
 - LMB: Shoot.
 - ESC: Pause menu.
- Intuitive Camera (2 points)
 - Orbital camera:
 - Left/Right arrow keys: Rotate camera horizontally around the level.
 - Up/Down arrow keys: Rotate camera vertically around the level.
 - Mouse scroll wheel: Change the distance to the level (zoom).
 - 1: Render the geometry pass using lines to show the wireframes
 - 2: Render the bullet collision shapes. - 3: Render SSAO map
- Illumination model (2 points) - 4: Render blurred SSAO map
 - Sun: Directional light. - 5: Render shadow map
- Textures (2 points): Tanks, Terrain
- Moving Objects (2 points): Tanks, Projectiles
- Documentation (1 points)
- Adjustable Parameters (1 point):
 - Camera: assets/settings/camera.ini
 - Window: assets/settings/window.ini

Optional:

- Advanced Gameplay (5 points):
 - Two types of enemies.
- Collision Detection (Basic Physics) (6 points):
 - Tanks can't drive through each and move along the uneven terrain.
- Advanced Physics (4 points):
 - Projectiles are physically simulated instead of using raycasts for hitscans.
 - The tanks are simulated with raycast vehicles.
- Heads-up Display (4 points):
 - Cursor position
 - Remaining health points

Effects:

Lighting:

- **Shadow Map with PCF (16 Points)**
Objects cast shadows on the terrain and on each other based on a directional light source.
 - **Core implementation (based on the LearnOpenFL tutorial):**
Reference: [LearnOpenGL - Shadow Mapping](#)

The scene is rendered twice. First from the light's point of view into a depth texture. Then in the main pass, each pixel checks whether it can "see" the light or is blocked by something. A small bias value prevents shadow acne artifacts caused by limited depth texture precision.

- **Extensions beyond the tutorial:**
 - Dynamic frustum:
Reference: <https://gamedev.stackexchange.com>
Instead of a hardcoded frustum for the depth mapping, the frustum is calculated dynamically from the camera frustum starting in normalized device space.

Advanced Modelling:

- **CPU Particle System (8 Points)**

40 particles burst out of the barrels tank, if a tank fires. The particles fly outward in a cone shape, slow down, fall due to gravity, and fade out as they die. The effect looks like a small orange muzzle flash.
- **Core implementation (based on the LearnOpenGL tutorial):**

Reference: [LearnOpenGL - Particles \(2D Game\)](#)
500 particles are always kept in a fixed array. When a shot is fired, 40 dead particles are reused and given a random direction, speed and lifetime. On each frame, every live particle moves forward and gets pulled down by gravity. Its alpha value fades from 1 to 0 as its lifetime runs out.
- **Extensions beyond the tutorial:**
 - GPU instancing:
Reference: [LearnOpenGL - Instancing](#)
All live particles are uploaded to the GPU in one buffer update and drawn in a single draw call, instead of one draw call per particle.
 - Billboarding:
Reference: [OpenGL-Tutorial - Billboards](#)
Each particle is a flat quad that always faces the camera. The vertex shader uses the camera's right and up vectors to orient the quad in 3D space.

Terrain:

- **Tessellation from Height Map (12 Points)**

The terrain is rendered as a flat grid of quad patches. Each patch is split into smaller triangles by the GPU, and a height map texture is sampled to displace each vertex up or down. This gives the terrain its bumpy shape without a heavy CPU side mesh.
- **Core implementation (based on the LearnOpenGL tutorial):**

Reference: [LearnOpenGL - Tessellation](#)
The pipeline has four shader stages: vertex shader, tessellation control shader (TCS), tessellation evaluation shader (TES) and fragment shader. The TCS decides how many triangles each patch gets. The TES interpolates the positions over the patch and examines the height map to displace each vertex on the vertical direction.
- **Extensions beyond the tutorial:**
 - Smooth normals via finite differences:
Reference: [Stack Overflow – How to calculate normals in a terrain...?](#)
Instead of using flat patch normals, the TES samples the height map at four neighboring texels and computes a smooth normal from the height differences. The approach is adapted from the SO reference: the axis order is changed from Z-up to Y-up to match OpenGL, and the sampling offset is normalized to the texture resolution (1.0 / 512.0) instead of a fixed world-space value.

Animation:

- **Hierarchical Animation (4 Points)**

The enemy turrets copy the position of the tanks and apply an offset to stay attached while moving. When the player is in sight, the turrets are rotated over time to aim at the player, otherwise they are rotated back to the forward direction. For this, a local turret rotation is stored and applied atop of the tank rotation.

Shading:

- **Physically Based Shading (6 Points)**

Albedo, ambient occlusion, metallic, roughness and normal maps are used per material to mimic light in a physically plausible way and achieve a more realistic look.

- **Core implementation (based on the LearnOpenGL tutorial):**

Reference: [LearnOpenGL - PBR/Lighting](#)

The shader implements Cook-Torrance BRDF, Fresnel-Schlick approximation, Trowbridge-Reitz GGX and Smith's Schlick-GGX using textures as input.

- **Extensions beyond the tutorial:**

- G-Buffer:

The shader is implemented using buffers for albedo, normals and ambient occlusion, roughness and metallic combined. For this, the three textures were merged into a single image utilizing the RGB channels and then exported as part of the GLTF 2.0 file format. The buffers are first populated in the geometry stage and later processed in the lighting stage.

- Single directional light

Instead of four point lights, a single directional light representing the sun was used, dropping the attenuation.

- Shadow map/SSAO

The values produced by the Shadow Mapping and the SSAO stage are applied to the direct lighting result and the ambient term.

Post Processing:

- **Ambient Occlusion (12 Points)**

Corners or tight spaces receive less ambient light than open areas. This makes the scene look more three-dimensional without calculating real global lighting.

- **Core implementation (based on the LearnOpenGL tutorial):**

Reference: [LearnOpenGL - SSAO](#)

The renderer runs in four steps. The first step is to render the scene into a G-buffer, which contains the position and normal of each pixel. The SSAO pass then samples 64 random points around each pixel to determine how many of them are occluded by nearby geometry. The result is a grayscale image with dark representing occluded areas. The noise is blurred with a 4x4 blur. Lastly the lighting pass multiplies the ambient light with the occlusion value.

- **Extensions beyond the tutorial:**

- Background guard:

When pixels don't have any shape, they return to full ambient occlusion (no darkening) right away. Samples that land on the background are also skipped. That way, the sky wouldn't darken shapes near the edges of the screen as it should.

- Screen bounds check:

Samples that are outside the screen are skipped so that data that is not stated is not read.