3D Game Engine Planning

Game Overview

Duck Hunt 3D is a reimagining of the classic arcade shooter where players use mouse-based aiming to shoot ducks flying through a 3D forest environment. Player will have to face rounds, which get progressively hard where they have to shoot at waves of ducks that fly around. With limited ammunition (10 shots per round), players must hit at least 7 ducks to advance. The game has low-poly 3D graphics with physically-based rendering. It will be built on a custom game engine with deferred rendering, the project demonstrates both technical graphics programming and solid gameplay design.

Core Gameplay

Players use their mouse to aim a crosshair and left-click to shoot. Each click fires a ray from the camera through the cursor position, which will be tested for collisions with duck bounding boxes in 3D space. Ducks spawn from random positions in a 180-degree frontal view and fly in parabolic arcs/ pre-determined paths across the screen for approximately 2-5 seconds before escaping. Hit ducks play a death animation (falling, fading and some effects maybe) and award points to the player. Players have exactly 10 bullets per round with no reload mechanic (for now). The round ends when all 10 ducks have been spawned and either shot or escaped, or when the player runs out of ammunition. Success requires hitting at least 7 out of 10 ducks to progress to the next round.

There are 5 rounds with increasing difficulty. Round 1 serves as a tutorial with slow-moving ducks (1.0x speed) spawning every 2.5 seconds. Each subsequent round increases duck flight speed by X% and reduces spawn intervals slightly. Players start each round with 10 bullets. Scoring is straightforward: 100 points per duck hit, with optional bonus for hitting more than 7 ducks in a round. The game tracks the high score only for a single session (no persistent data). Failing to hit 7 ducks in any round results in game over, requiring a full restart from Round 1 (aka fresh start).

Visual Design

The game features a low-poly art style using Synty asset packs, with models ranging from 500-2000 triangles for optimal performance. The 3D environment includes a ground plane, scattered trees, and a skybox (optional/low priority). The rendering system uses deferred PBR with a G-buffer storing position, normal, albedo, metallic, and roughness data (ideally), followed by a lighting pass implementing BRDF for realistic material appearance. The camera is fixed at position looking forward with fixed field of view, providing a first-person perspective similar to

the original Duck Hunt. Visual effects might include a simple muzzle flash on shooting, hit effects when ducks are struck, and optional feather particles on duck death (post-beta). The UI overlay displays score (bottom-right), round number (bottom-left), and remaining ammunition as visual bullet indicators plus the number of ducks hit (bottom-center), with a crosshair following the mouse cursor

Technical Implementation

The game is built on a custom engine using OpenGL with deferred rendering architecture(ideal). The Entity Component System manages 6 core component types: Transform (position,rotation,scale), MeshRenderer (visual representation), Movement (velocity based motion), Collider (AABB for hit detection), DuckAI (behavior state machine, pretty bare), and Lifetime (auto-destruction timer). Systems are - MovementSystem applies velocities, DuckSystem manages flight paths and state transitions (fake animations), and RenderSystem handles all drawing. Shooting is implemented via ray casting from camera position through the mouse cursor in screen space, transformed to world space, then tested against all duck AABBs using the slab method for ray-box intersection. The Resource Manager/ Asset Manager to cache all loaded assets (models, textures, shaders, materials) probably uses tinyobjloader for .obj files exported from Synty packs via Blender, with textures loaded via stb_image. Audio (post-beta) uses OpenAL for simple 2D sound effects triggered on game events.

Engine Architecture

Architecture Overview

Follow a component-based architecture with a central "Engine" class managing all major subsystems. At the highest level, the Engine owns and coordinates the Renderer, EntityManager, ResourceManager, InputManager, AudioManager, CollisionSystem, and GameStateManager. These systems communicate through interfaces but mostly independent, which perfect for our team to work in parallel to setup the managers. We use a traditional game loop pattern: process input, update all systems with delta time, then render the scene, swap buffers and repeat. The separation of dependency means that someone can work on rendering without breaking gameplay logic, while some is working on the gameplay without touching rendering code.

Rendering System Architecture

The rendering system is the main goal of this project and should be our main focus (we can implement whatever we want), implementing deferred rendering with physically-based rendering (PBR) as the primary goal. The ideal implementation uses a G-buffer with some textures (4 max) storing position, normals, albedo/metallic, and roughness data, followed by a lighting pass that uses Cook-Torrance BRDF. I think Chris and I, know a bit about PBR and i think this is the best source to read about it. Also if you thing this a lot, the learnOpenGL does 80% of the rendering we need to do for our game, so its not that difficult. If time permits after beta we can enhance our rendering system with shadow mapping using a depth pass from the light's perspective, or do image-based lighting (IBL) using the skybox as an environment map for more realistic reflections and ambient lighting (not that much work but would add a lot of visual detail). However, we should have a fallback plan always: if deferred rendering proves too time-consuming or problematic, we can leave it at forward rendering with Blinn-Phong lighting like dirk engine which is still not bad still looks professional.

The rendering architecture has three coordinated passes—geometry pass for writing to G-buffer, lighting pass to computing final colors, and forward pass for transparency and UI (not real rendering). Post-processing effects like bloom or HDR tonemapping can also be dont but are considered low-priority, leave it for the final week if the core rendering pipeline is stable and performant.

resources
https://www.pbrt.org/
https://learnopengl.com/Introduction
https://resources.mpi-inf.mpg.de/departments/d4/teaching/ws200708/cg/slides/CG07-Brdf+Texture.pdf

Entity Component System Design

The ECS uses a simple pointer-based approach same as the 2D engine we made in class. Each Entity object contains pointers to its components: Transform, MeshRenderer, Movement, Collider, DuckAI, and Lifetime. The EntityManager maintains all of the Entities and assigns unique IDs. When code needs to update all entities with specific components, it asks EntityManager for a filtered list, like "give me all entities that have both Transform and Movement components."

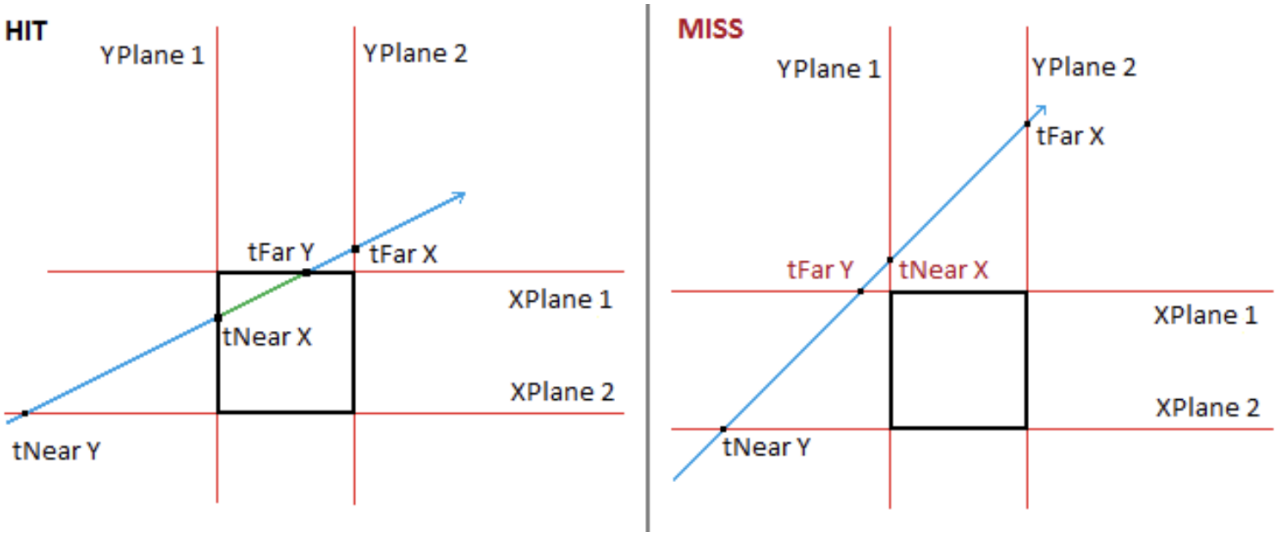
For system we just iterates through its relevant entities and updates them: MovementSystem adds velocity to position, DuckSystem updates flight paths, LifetimeSystem decrements timers and destroys expired entities. The RenderSystem reads Transform and MeshRenderer data to submit draw calls. We can add more components or systems if required

Resource Management Strategy

The ResourceManager is for central asset cache using string-based lookup. When we request a model with the name "duck", the manager first checks if it's already loaded in the models map. If found, it returns the cached pointer otherwise, it loads the file from disk and stores it in the map, then returns the pointer. Same pattern applies to textures, shaders, and materials. All resources are cleaned up in the manager's destructor, stop memory leaks when the engine shuts down unless you want them.

Physics and Collision Implementation

The collision system is pure raycasting with NO physics simulation, which we don't need for our game. When the player clicks to shoot, the InputManager provides the mouse cursor's screen-space coordinates. The CollisionSystem transforms to get the world space position: input position ->NDC->clip space->camera space->world space, producing a Ray with origin at the camera position and direction pointing through the cursor into the scene. This ray is then tested against every entity that has a Collider component.



For collision checking we use **Cyrus-Beck clipping** for raycast-AABB which gives us T1(closer) and T2 where the rays collides with the box, this is a good resource for implementing and understanding Cyrus-Beck clipping. The ray-AABB intersection uses the slab method, testing the ray against each of the three axis-aligned slabs (X, Y, Z) and finding where it enters and exits each. If the ray is inside all three slabs simultaneously, there's a hit. We jst need to track the closest hit distance and returns which entity was hit. For duck death physics, there's no actual physics engine when a duck dies, it simply gets a downward velocity applied each frame to simulate falling via gravity. This "fake gravity" is sufficient for the falling death animation.

Game State and Flow Management

The GameStateManager maintains the current state (Playing, RoundEnd, GameOver, GameComplete) and all gameplay display variables like score, round number, ammunition count, and ducks hit. State transitions happen based on game events: when all ducks are spawned and either shot or escaped, transition to RoundEnd or if the player hit enough ducks, advance to the next round or GameComplete, otherwise go to GameOver. In Playing state, the duck spawning timer counts down and creates new ducks at intervals, and player input is processed for shooting. In RoundEnd state, no updates occur and the system waits for player input to continue. The manager also tracks scoring with methods like onDuckHit and onDuckMissed that update the score and bonus points.

Input System Design

During the game loop's input processing phase, the Engine queries InputManager and dispatches actions: if left mouse button just pressed and game state is Playing, call the shoot method; if SPACE pressed and state is GameOver, call restart method. Basically the InputManager knows nothing about game logicit, purely an input state container.

Audio System

The AudioManager wraps OpenAL for simple 2D sound playback. We need audio for background music and sound effects for shooting, hitting ducks, spawning duck quack, etc. Sound loading reads WAV files from disk, creates OpenAL buffers, and stores buffer IDs for lookup by name. Audio is optional for beta and can be stubbed out or disabled entirely. Also Just 2D audio is sufficient.

Model Export Workflow

This might be one of the harder part of the engine, since Synty assets come as FBX files incompatible with the simple OBJ loader, fist we need to convert models before runtime. Opens each required model in Blender, imports the FBX with default settings, and inspects the mesh. Merged into one object for simplicity since animation isn't implemented if the model has seperate meshes for wing or head. Rotated and scaled if necessary (this was an issue in SceneKit where -Y was up) match it with engine's coordinate system. Export with "Include UVs", "Write Materials", and "Triangulate Faces" options checked, forward axis set to -Z and up axis to Y. Then we will have an OBJ file with the mesh data and an MLT file with the material mapping not the actual texture, but i think synty packs everything into a single png (so grab that aswell). For load the obj and mtl we can just use this. We just need to make sure we figure out the correct configurations for this

Game Loop and Timing

Main game loop where we pool input events, call the game update systems and handle round progression. Then the render phase executes all rendering passes and draws the UI

Duck Spawning and Lifecycle

The GameStateManager maintains a spawn timer that decrements each frame by delta time. When the timer reaches zero andall10 ducks have not been spawned this round, it calls spawnDuck and resets the timer. Creates a new entity adds Transform, MeshRenderer, Movement, DuckAI with flight path parameters, Collider with AABB bounds, and adds Lifetime set to the flight duration.

Then each frame, the DuckSystem iterates all entities with DuckAI, checks their state, and if Flying, interpolates position along path. When t reaches 1.0, the duck escaped—the system destroys the entity and notifies GameStateManager of the miss. When a duck is hit, the shoot method changes its state to Dead, and does executes the death state, and shortens its Lifetime to 1 second after which it is removed.

UI Rendering Strategy

The UI overlay displays score (bottom-right), round number (bottom-left), and remaining ammunition as visual bullet indicators plus the number of ducks hit (bottom-center), with a

crosshair following the mouse cursor. All UI rendering happens in the forward pass after 3D rendering completes. This can be pretty basic.

Task Planning

So we have three deadline one is the GDD due on 1st (he said in class today), then beta due on Nov 18th and final presentation December 2nd (dk if we present in class).

Task to complete before starting

- Plan the GDD
- Scope out tasks
- Address any risks
- Task ownership (?)
- LearnOpenGL (1-4) get to render to a window atleast
- Setup jira and github (let me make the repo :))
- Setup OpenGL GLAD GLFW etc get it compiling

Before Beta Goal

Goal: **Prove the shit works, what "works" means:**

- Core gameplay loop is functional (not polished)
- Technical systems are stable (not optimized)
- Playable from start to finish once (without crash)

Non-Negotiable

Gameplay:

- Duck spawns at random position
- Duck flies across screen (parabolic arc or path idk)
- Click mouse to shoot
- Ray cast detects if you hit duck
- Duck disappears when hit
- Score increases when hit (+100)
- Ammo decreases when shoot (10 total)
- Round ends when 10 ducks spawned OR ammo = 0
- "You passed" or "Game Over" screen or console
- Can restart game (some key)

Rendering:

- 3D duck model loads and displays (get a pipeline working at least)
- Basic lighting (forward or deferred - either is fine)
- Textured models (duck, ground, trees)
- Camera works (fixed position)
- Skybox (makes it look less empty but might be overkill)

UI:

- Score displayed (top-left)
- Ammo count
- Round number (top-center)
- Crosshair follows mouse
- Round end/ death overlay

Technical:

- No crashes during normal gameplay
- -Assets load correctly (not upside down)
- ECS manages entities
- Basic game state management

Final

Goal: **Portfolio-worthy non synty asset flip, feels complete, what "complete" means:**

- Game is fun and can be played
- Systems are polished and optimized
- Looks professional
- No obvious bugs or rough edges and 30-60 FPS

Non-Negotiable

Gameplay:

- 5 full rounds with progression
- Duck speed increases each round
- Round transition screens (just overlay)
- Proper game over screen with final score
- Duck death animation (falls and fades)
- 7/10 ducks logic working perfectly

Visual Polish:

- Deferred PBR rendering (if not in beta)
- Particle effects (can be faked by billboards or lights):
  - Muzzle flash when shooting
  - Feather explosion when duck hit

Audio (completely optional but ask Charles):

- Gunshot
- Duck quack
- Hit confirmation
- Miss sound

Technical Excellence:

- Optimized rendering (consistent 60 FPS)
- -No memory leaks
- No crashes while presenting in front of the class
- Clean, documented code and professional build/release