Ubisoft NEXT Programming 2025

Jennifer Kim

YouTube Link: https://youtu.be/ENdJ8lNCyjQ

Gameplay Overview

The game follows the premise of mini-golf with a 3D third-person view, physics-based mechanics, and puzzle/maze elements while supporting both single-player and two-player modes.

Objective

The goal is to navigate through the 3D maze and get the ball to the destination with the fewest possible movements. In 2-player mode, players take turns shooting the ball. The player who reaches the destination faster wins. Alternatively, if one of the players dies, the surviving player is declared the winner.

Game Controls

- **W**: Move Camera Forwards
- **S**: Move Camera Backwards
- **A**: Look Left
- **D**: Look Right
- **SPACE**: Hold to control intensity, release to shoot the ball in the forward direction of the view.
- **LShift/RShift**: Switch the type of golf club (PUTTER or DRIVER).

Game Features

- Similar to a real-life mini-golf, a turn starts when the ball is shot from its resting position, and the turn ends when it returns to the resting position.
- The ball can only be shot from the resting position using two types of golf clubs:
  - **PUTTER**:
    - Rolls the ball on the ground in the forward direction.
    - The yellow gauge bar controls the force, with fuller bars exerting greater force.
  - **DRIVER**:
    - Shoots the ball in a projectile motion.
    - The gauge bar controls the angle of the projectile:
      - Fuller bars indicate higher angles (ranging from 10° to 80°).
      - The longest distance is achieved at approximately 45°, marked by a white line on the gauge bar.
- Each block is bounded with AABB and when the ball collides with the block from the side ball bounces in the opposite direction. If the ball is on the block, the block applies friction to the ball in the opposite direction. Different types of blocks interact with the player differently:
  - **Green Block:** Basic block representing grass

    - **Yellow Block**: High-friction block representing sand
    - **Pink/Blue Blocks**: Portals that teleport the ball to the corresponding coloured block:
        - Pink portals have a cooldown of 8.0 seconds (20.0 seconds in multiplayer)
        - Blue portals have a cooldown of 12.0 seconds (25.0 seconds in multiplayer)
    - **Moving Block**: Blocks that move within a predefined range
    - **Goal Block**: The game ends, and notifies the player that they won :)
    - **Collision Between Balls** (2-player mode): When two balls collide, they bounce off each other in an elastic collision motion.
- If a player falls to the ground, they lose one life. Each player starts with 3 lives, and the game ends if a player loses all their lives.
- Particle System to indicate the goal, as well as player's collision to the wall

Challenges and Solution

Implementing Multiplayer Feature

The hardest challenge for me when developing this game was implementing the multiplayer feature. The primary issue was that players have multiple lives causing them to simultaneously disappear from the scene and reappear back into the game. And this issue was amplified as the initial design assumed for only one player.

Odd Behaviors Observed:

1. The player starts at the ground and does not respawn back at their turn:
    - A player could die during the other player's turn, when their being collided with and pushed off a block.
2. Turn mechanics being violated:
    - A player's turn could start even while the ball was still moving, allowing them to shoot again before the ball returned to a resting state.

Solution

There must be a more elegant and less messy way to resolve this issue, but my solution was to divide it into two distinct parts: 1. one ball exists in the scene and 2. two balls exist in the scene.

Additionally, instead of depending on the movement of the current player to end the scene, I modified it to keep track of the states of both balls to ensure that both of the balls are at resting position, before switching turns.

This solution was effective, especially as I knew the scene works perfectly when there is one player as it was preimplement before adding one more player. And I can strictly focus on the logic of the scene when expecting 2 balls.

Key Engine Features

3D Graphics Pipeline / in Graphics Folder (mainly Renderer.cpp) **and Vector Math** / in Math Folder

The engine supports math for Vec2, Vec3, Mat3x3, and Mat4x4, which is essential in creating the graphics pipeline. All the transformations I used are stored in the Mat4.h.

When individual mesh is drawn to the screen, it undergoes the following transformations:

1. **Affine Transformations**: Scale, Translate, and Rotate coordinates in object space.
2. **Backface Culling**: Using the direction of the camera forward and position, skip coordinates at the back
3. **View Matrix Transformation**: Converts coordinates into camera view space.
4. **Perspective Transformation**: Changes coordinates into a frustum, making objects appear larger when closer.
5. **Orthographic Transformation**: Converts coordinates into a normalized clip space between -1 and 1 (NDC).
6. **Frustum Culling**: Skip coordinates outside the range of -1 to 1
7. **Viewport Transformation**: Scale and transform the points between -1 to 1 to fit the actual screen
8. **Triangle Rasterization**: Using ambient + diffuse lighting to determine the colour of each triangle and colour the triangle

Rasterization / in Graphics/Renderer.cpp

To rasterize the triangle I implemented two methods:

1. **Scanline Method**:
   - Choose the longest side of the triangle along the y-axis as side 1.
   - Interpolates values while moving down the y-axis, drawing lines along the way.
2. **Pineda's Algorithm**(https://www.cs.drexel.edu/~deb39/Classes/Papers/comp175-06-pineda.pdf):
   - Calculates all possible points within the bounding box of the triangle.
   - Determines whether each point lies inside the triangle by evaluating edge functions.
   - Initially implemented wanting to perform parallel but wasn't able to (something to work on in the future!!)

Both algorithms perform rasterization without making direct OpenGL calls, using the App::DrawLine function by the provided API.

ECS and Memory Management / in ECS Folder

The *SceneGolf* class manages all the systems (or actions) in the game. An instance of this class is stored in the GameTest.cpp file. *SceneGolf* inherits from *Scene*, which contains an instance of the *EntityManager* class as a protected variable.

The *EntityManager* class is responsible for managing all the data within the game through:

1. **MemoryPool**:
   - Holds static data of component classes as fixed-size vector
     - Each index represents an entity
   - Singleton design pattern, ensuring only one memory pool in the scene
   - Can only be accessed by the SceneGolf class through the *EntityManager* class
2. **Bounding Volume Hierarchy (BVH)**:
   - The *EntityManager* holds a std::unique_pointer to the BVH.
   - Stores collision data in an ordered manner to optimize time during collision detection.

This design in the ECS is cache-friendly and avoids the usage of raw pointers. The only explicit heap allocation using a pointer is for the *BVH*. This approach resolves the circular dependency between the *EntityManager* and *BVH* classes, which arose when storing *BVH* on the stack inside *EntityManager*.

Memory Fragmentation Avoidance

The memory pool avoids this issue as the **size is fixed**, and each entity has a **uniform size** as all the components are preallocated (some entities may not need it but ensures uniform size). And the pool **reuses the spot** for the

removed entity.

However, BVH, cannot completely avoid fragmentation since it is implemented dynamically on the heap. To minimize this issue I am using the concept of the memory pool:

1. At initialization, the size of the nodes vector in the **BVH is set to the maximum entity** value defined for the memory pool. While the size of a binary tree could theoretically reach 2n, the maximum number of entities during gameplay keeps this value reasonable.
2. Each **BVHNode is the same size** and deleted positions in the vector are **reused** before adding new nodes, reducing frequent resizing of the vector.

Dynamic Bounding Volume Hierarchy (BVH) / in Physics/BVH

As I mentioned above, the game engine uses an Axis Aligned Bounding Box BVH for collision detection rather than looping through the entire dataset. I am building the tree bottom-up and dividing it by taking the median of the longest axis. Considering the game entities mostly being static and evenly spaced out, this method avoids the worst-case scenario of the height of the tree being n length. Therefore this approach reduces the complexity of collision detection, as the height of the tree is the average of log n.

- Instead of an O(n) complexity for checking collisions for a single entity, the implemented BVH tree achieves an O(log n) complexity.
- This optimization significantly improves performance, especially for large datasets, by organizing collision data hierarchically and reducing unnecessary comparisons.

However, the additional cost (other than extra space) comes from adding and removing entities from the tree. When adding or removing a node, the values for all of its parent nodes need to be updated, costing an additional O(log n).