Name: Harsh Naidu
Student Number: S240420042

Email ID: H.Naidu2@newcastle.ac.uk

Youtube Video Link: https://youtu.be/jM27XU_8HVQ

Player Movement: Yes

- Player can move:
  - Player can move using [W],[A], [S], [D] keys and Jump using [Space] key.
- Player can rotate:
  - Player can rotate using W],[A], [S], [D] keys.
- Player uses forces/impulses for movement:
  - Player uses forces for forward, backward, left, and right movements.
  - Player uses Linear Impulse for jump.
- Player uses torque/impulses for rotation:
  - I have used forces for rotation as well, because I have implemented a third person camera that you can rotate the camera around the player, so using [W], [A], [S], [D] keys for movement also rotates the player.

Collisions: Yes

- Sphere/Sphere: Player can collide with the ball trapped within the OBB walls. It's towards the South-East of the initial player position.
- Sphere/AABB: Player vs Purple (AABB) cubes in the world. Player has a sphere collider attached to it. Also, the sphere and the floor (AABB volume).
- AABB/AABB: Black Cube (AABB) at the front of the door puzzle, initially placed in the air drops on the floor (AABB) and bounces.

Basic Extra Collisions: Yes

- Something vs Plane: Plane with the default (doge) texture colliding with the AABB towards north-east of the player's initial position.

- OBB vs Sphere: Sphere is surrounded by OBB walls towards south-east of the player's initial positions.

Note: OBB's have been given a **purple texture** to make it easier to spot them.

- Raycast vs world used somewhere in the game:

    - Player uses raycast to check the jump distance.
    - Enemy uses raycast to detect and chase the player.
    - Kittens use raycast to detect and follow the player.

Collision Resolution: Yes

- Projection method used: Projection method has been used for basic collision resolution. It ensures that two objects are not overlapping. Check the ImpulseResolveCollision method for it.

- Impulse method used: Impulse method has been used for basic collision resolution. It calculates and applies impulses to resolve the collision velocity and rotational effects. Check the ImpulseResolveCollision method for it.

- Multiple coefficients of of restitution: Check the ImpulseResolveCollision method, the hardcoded cRestitution value has been changed with,

    float cRestitution = PhysA->GetElasticity() * PhysB->GetElasticity();

    - Black cube's elasticity changed to 1.0f.
    - Capsule's elasticity changed to 0.5f.
    - Plane's elasticity changed to 0.7f. There are two planes both with default (doge texture)

- Penalty method used somewhere:

    - Penalty method used to resolve collision between the ball (sphere volume) and cube wall of OBB's. Check the ResolveSpringCollision method in PhysicsSystem.cpp. Have a look at the BasicCollisionDetection method, how I have applied the ResolveSpringCollision method to the only sphere.

Stateful Behaviour: Yes

- Simple menu implemented:
    - Press [C] to start the game.
    - Press [P] to pause the game.
- Stateful obstacles in level:
    - Enemy, the moving door, and everything in the game pauses when you pause the game.
- Multiple different obstacle types:
    - A maze with an angry goose
    - Door puzzle, the door is constantly moving. You have to just time your movement and wait for the door to go up and then you can go in.
    - Flying stairs
    - Straight bridge puzzle
    - Circular bridge puzzle
    - Flying wall with a springy sphere in it. Move the springy sphere to get the bonus

Gameplay Effects: Yes

- Player can collect bonuses: Player can collect kitten heads as bonuses. They are placed around different puzzles in the game.
- Player can win game: To win the game the player need to collect 7 bonuses and help the 3 kittens to reach the home. Total 10 points are required to win the game.
- Player can lose game: The game lasts for 5 minutes, there is a timer at the top-middle of the screen. If the player does not score 10 points in 5 minutes, the game over screen is displayed.
- Player shown final score: Player can see the final score when the game ends.

Advanced AI: Yes

- AI Opponent present: AI opponent is pressed in the game as an angry goose wandering the maze.
- State-based opponent AI: AI opponent wanders the maze, if the player is close to the angry goose it uses raycasting to chase and detect the player. If the player is too close to the angry goose the player will respawn back to it’s initial position. Check the StateGameObject.cpp and have a look at the way it has been implemented

- Behaviour Tree opponent AI: Not implemented

- AI uses raycasting: If the player is close to the angry goose it uses raycasting to chase and detect the player. If the player is too close to the angry goose the player will respawn back to its initial position.

- AI can collect bonuses: Yes, AI can collect bonuses. I have just used the OnCollisionBegin method for it.

- AI can try to avoid an obstacle: Yes, AI can avoid obstacles.
- AI can teleport/respawn if necessary: Not implemented

Pathfinding: Yes

- Grid-based pathfinding demonstrated: Yes, grid-based pathfinding has been implemented. Have a look at the SetupEnemyPath to see how I have implemented it.

- AI uses pathfinding: The AI uses this path to wander through the maze back and forth.

- AI can follow path: Yes, the path loaded by the SetupUpEnemyPath method has been passed to the AI's StateGameObject.cpp constructor which then the AI uses to follow the path using the state machine.

- NavMesh Pathfinding demonstrated: No, not exactly, but I tried and almost did everything to get it working. But, I couldn't get the start position and end position properly. I tried logging all the vertices to the console and calculating the start position and position using the centroid of the triangle, but the program wouldn't run. So, for now, I just passed some random positions to it and it logs in the console "No path found for Navmesh!".

Note: Please provide feedback on how I could have found the start position and end position for the NavMesh. As I used the same Navmesh asset provided to us in the project directory's assets folder.

Constraints: Yes

- Position constraint demonstrated: A straight bridge that leads from the flying stairs to the circular bridge. It uses 2 anchor points. Check the PositionBridgeConstraint.cpp method for it in the TutorialGame.cpp.

- An obstacle uses constraints. Two obstacles use constraints. The straight bridge and the circular bridge.

- Orientation constraint demonstrated: A circular bridge that starts at the end of the straight bridge. It has 4 anchor points and the constraint has been set up using the OrientationConstraint class.

Advanced Collision Resolution: Yes

- Penalty method used somewhere: Penalty method used to resolve collision between the ball (sphere volume) and cube wall of OBB's. Check the ResolveSpringCollision method in PhysicsSystem.cpp. Have a look at the BasicCollisionDetection method, how I have applied the ResolveSpringCollision method to the only sphere.

- Friction applied during resolution: Friction has been applied to the angry goose to slow it down after it collides with the maze walls. Have a look at the AddAngryGooseToWorld method in TutorialGame.cpp.

Advanced Collision Detection: Yes

- Capsule vs Sphere: Player (Sphere Volume) can collide with the capsule.

- Capsule vs AABB/OBB: Capsule with doge (default) texture, initially in the air drops onto the AABB cube. This is placed towards the north-east of the player's initial position.

- Capsule vs Capsule: Albeit not perfect the Capsule vs Capsule collision has been implemented towards the west of the player's initial position. Two capsules placed one above the other and then collide with each other.

- OBB vs OBB: OBB vs OBB collision has been implemented towards the right of the player's initial position. Again, the OBB's are purple, so shouldn't be difficult to spot it.

- Spatial Acceleration Structures used: Nope, not implemented.

Advanced Menu: 4 marks in total: Yes

User can select different game types: The player can select between two different game types, Time-based and Free-Roam (No timer). The player can toggle between the two game types using the [T] key.

Appropriate handling of menu state: Yes the menu has been appropriately handled in the UpdateGame method of the TutorialGame.cpp.

Networking:

- Networking has been implemented. I can receive packages on both the Server and Client side. The player movements are synced using the Player_Update package.

  To test the networking, just open the RunServerClient.bat file in the project directory. This will open two windows server and client. As of now, you can control the server player and watch it move in the client. Haven't done the same the other way around (client movements synced with the server).

Note: I found it very challenging to debug the networking. I did download WireShark to look at the packages being transferred from the server to the client and vice-versa. But, debugging both the client and server on the same machine was a little difficult when things went wrong.

Request for the next semester:

- Please conduct a bonus lecture where I can see you implement the networking and if things go wrong, how you debug them efficiently.
- If a lecture isn't possible, I can come in the after hours to learn about it. Couldn't get any time due to the pressure of this coursework.

Thanks for taking the time to grade my coursework. I have added some extra notes where I am expecting some feedback from you to improve. Please consider it.

—---End—--