700110 ACW Report

Practical physical based modelling and artificial intelligence assignment

Criteria 1.1

Newtonian physics will be applied when agents move, and to cannon balls firing/thrown through the scene.

Implementation details:

Physics engine module: This includes the code for simulating the physics of the objects in the scene, such as collision detection, numerical integration.

Cannon ball follows a projectile trajectory when launched from the tower or thrown on the terrain by an agent.

Muzzle velocity is provided to launch the cannon ball either from the top of the tower or via an agent on terrain. The projectile then moves under the influence of gravitational force, and experiences air drag during the course of its flight.

The agents can move across the map under the influence of the steering force acting on them.

All physical constants are maintained in the **`Config.h`** file.

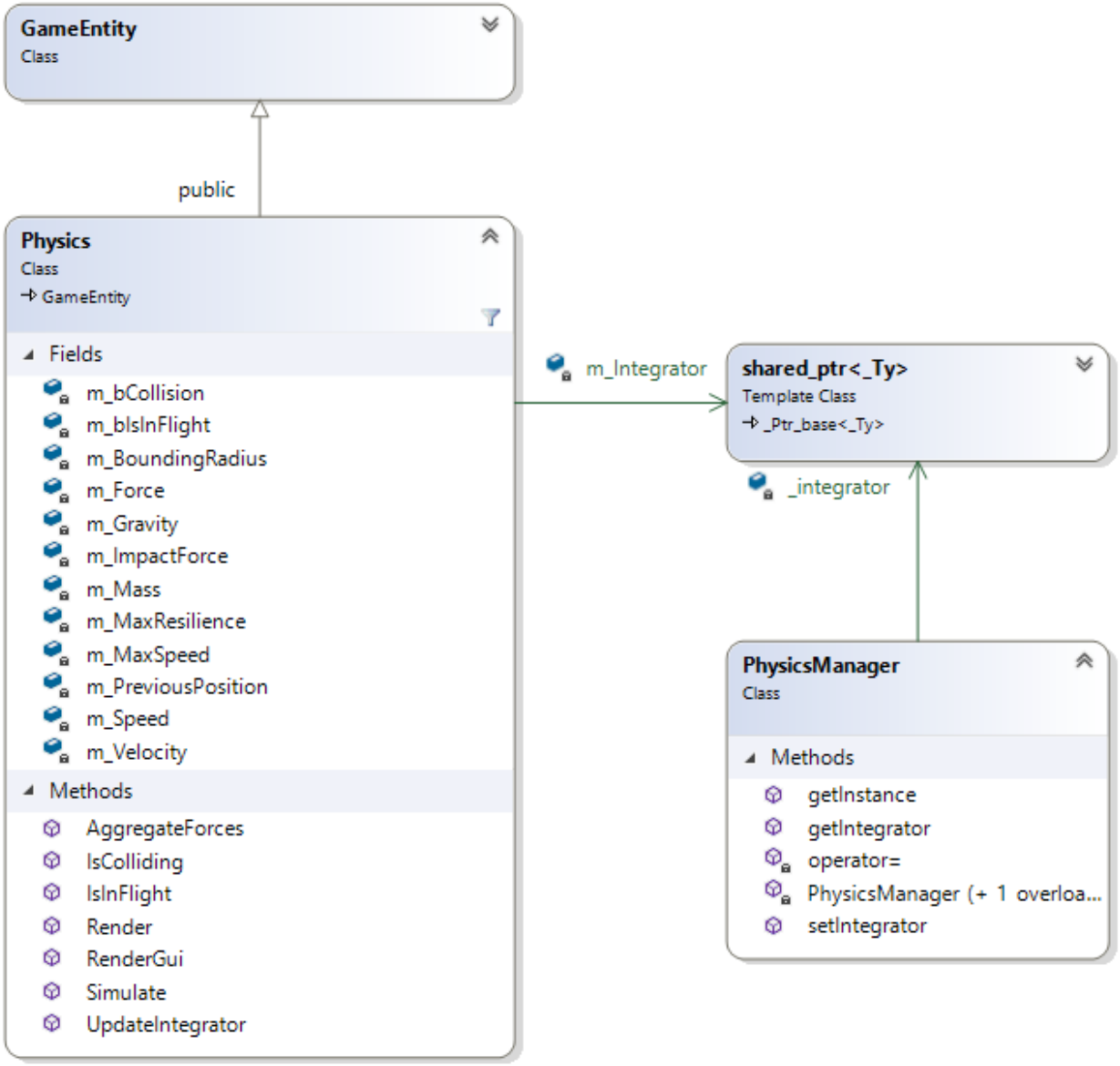
We define a **`Physics`** class which extends the **`GameEntity`** class to apply Newtonian physics to game entities.

`Physics` class has a function named **`AggregateForces()`** to reset all the forces in each frame, and resolve each force vector before calling the **`Stimulate()`** function to apply numerical integration.

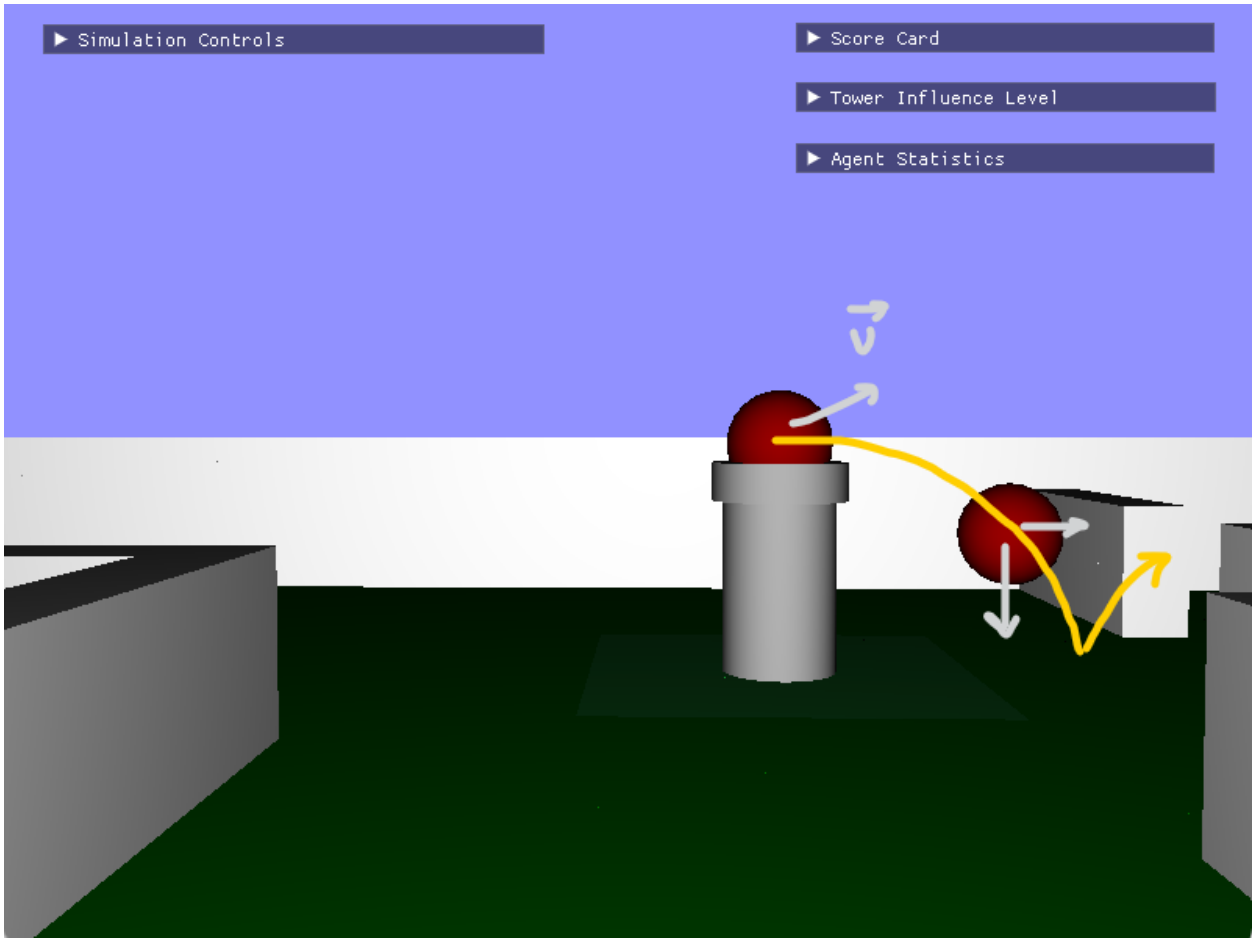
Physics class has floating point attributes to store MaxSpeed and MaxResilience (the maximum force an entity can produce) of the physical object, which allows the application to prevent instability.

Code evidence and screenshots:

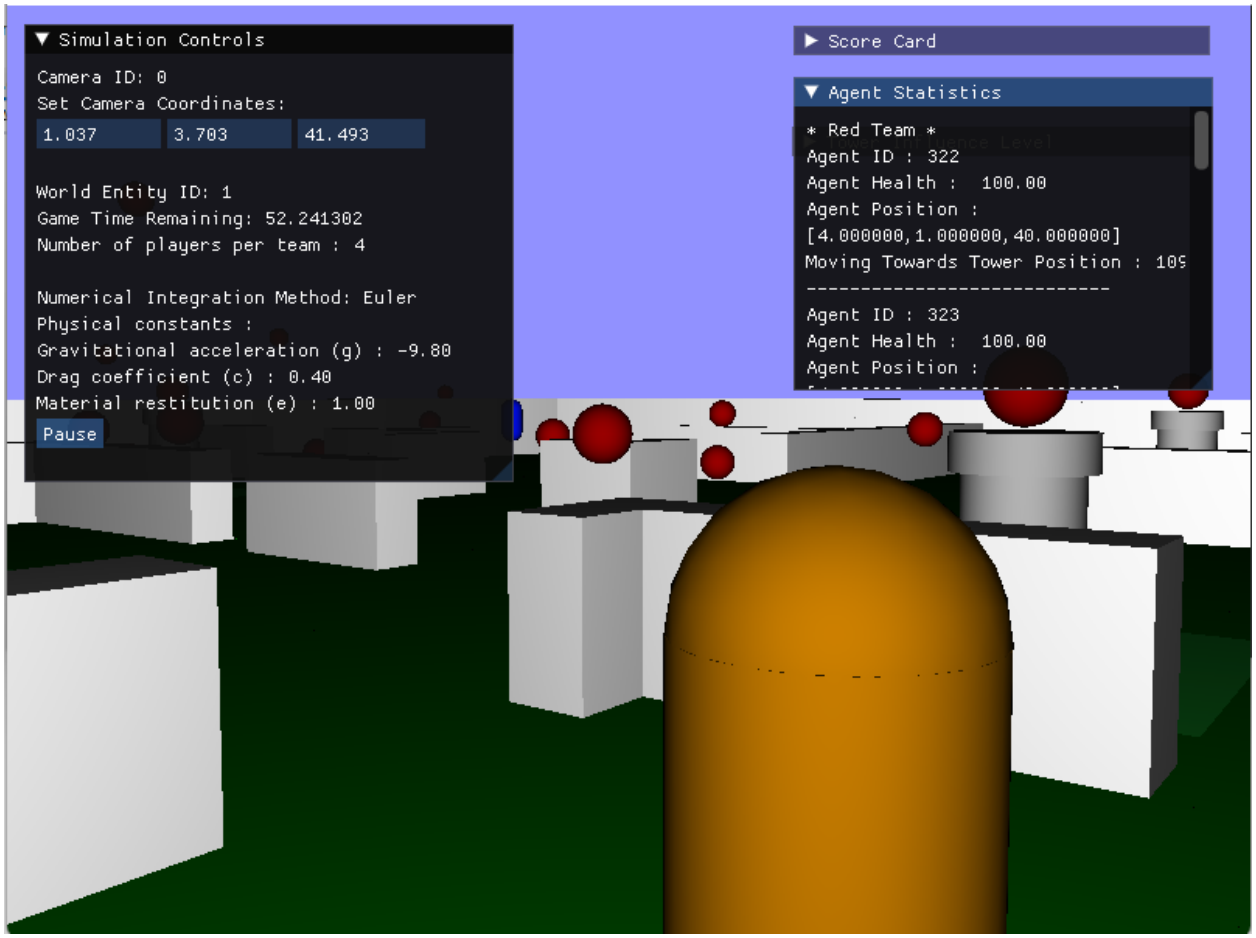
Filename: Physics Engine/Physics.h
Class names: Physics, PhysicsManager
Method names: Stimulate



** Please note that only key fields have been shown in the above class diagrams. Class constructors and other member functions are hidden.



Illustrating projectile motion in cannonballs.



Illustrating agents' motion on the field.

Reflection:

Newtonian physics applied accurately with pragmatic measures to prevent instability (i.e. capping velocities and forces) written in a reusable way.

Criteria 1.2

Numerical integration will be applied to all moving objects. In order to achieve full marks at least three methods of integration will be implemented, and it will be possible to change this at run time.

Implementation details:

Application has the following three methods for numerical integration, which can be switched at the run time using the menu:
1. Euler
2. RK4
3. Verlet

Error-controlled time stepping: In the implemented approach, the integrator uses a fixed time step, but the step size is adjusted dynamically based on the error in the solution. This can be done by comparing the solution obtained using the current time step with a reference solution, and adjusting the time step based on the difference.

App.cpp has **renderImGui( )** function which allows to switch integrator type based on a shared pointer which is created and maintained by a singleton class **PhysicsManager**.

The PhysicsManager class has a createIntegrator method which creates an Integrator object of the specified type and returns it as a **std::shared_ptr.**

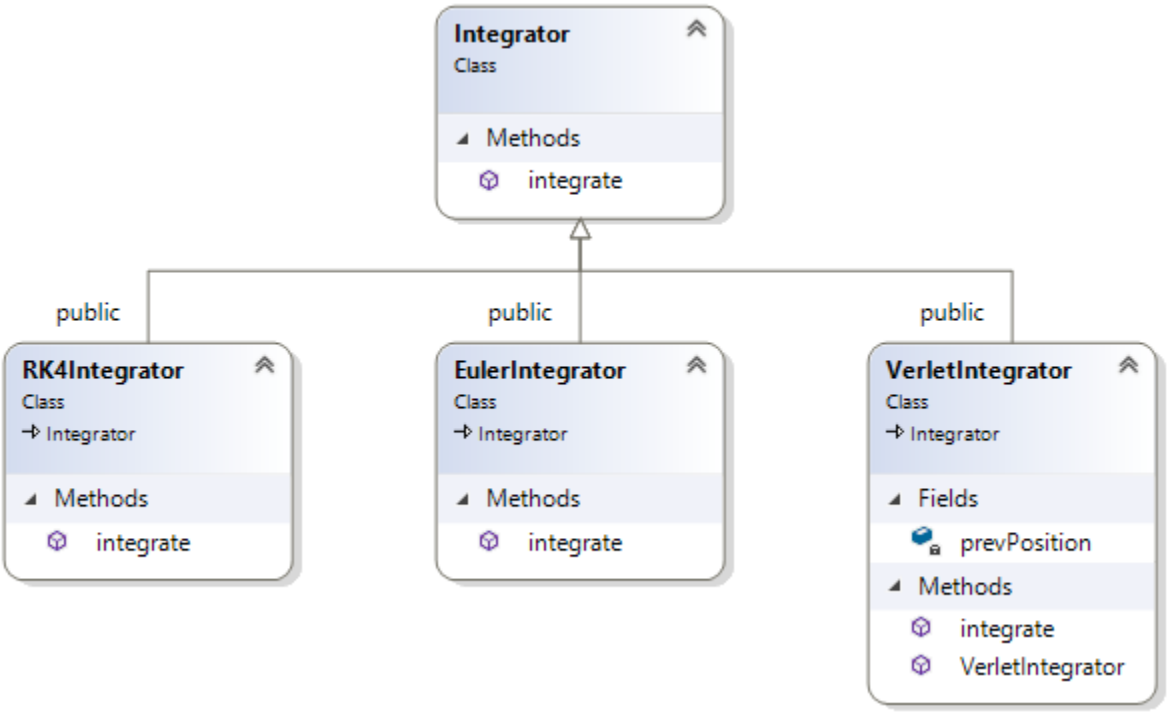
All entities extending physics class have a shared_ptr to an Integrator object. The **Update ( )** method calls the **Simulate( )** method of the Physics class passing the **dt** and the integrator shared_ptr.

By using std::shared_ptr the management of the memory of the Integrator objects is taken care of automatically, and it makes sure that the memory is not deleted before it is needed. This way, the Integrator objects are deleted when the shared_ptr goes out of scope, and it eliminates the need to manually delete the objects or to include a destructor in the Simulation class.

Code evidence and screenshots:

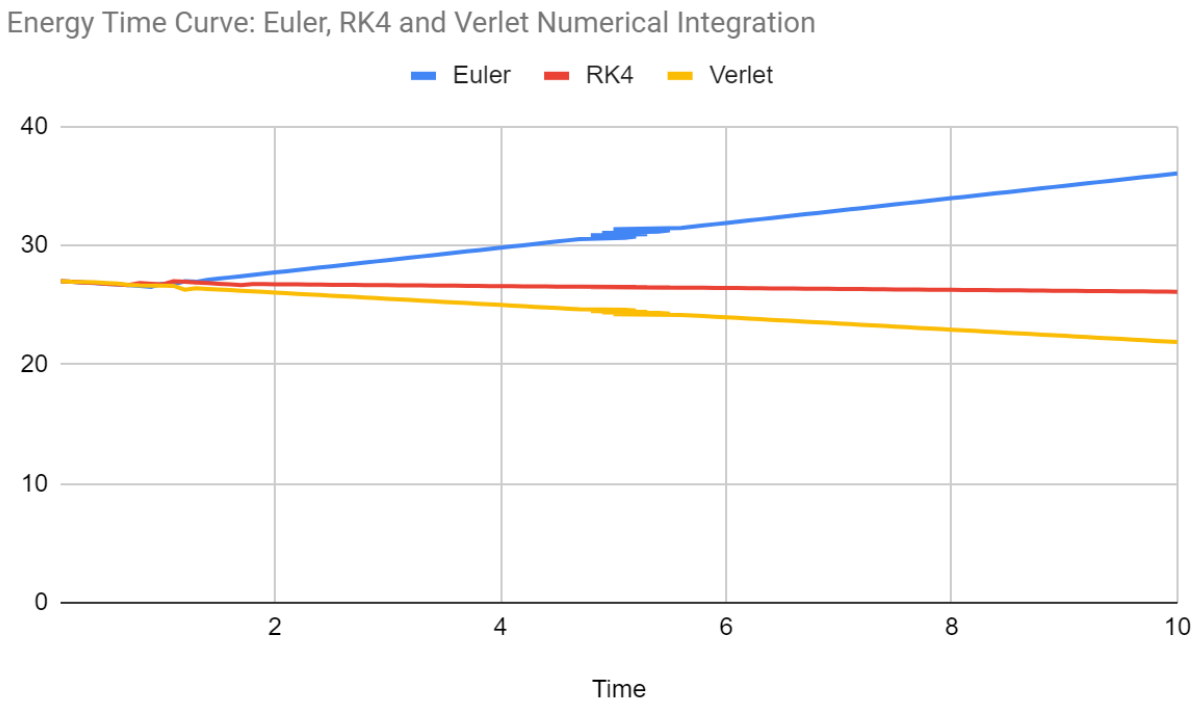
Filename: Physics Engine/Integrator.h
Class names: Integrator, RK4Integrator, EulerIntegrator, VerletIntegrator
Method names: integrate

UML Diagram: Integrator class and its subclasses.



To illustrate the efficiency of each method, we can analyse the projectile's trajectory and system’s energy over time.

We’ve the following Energy v. Time plot comparing energy dissipated during the simulation.



Measuring conditions:
The coordinates were captured for a projectile of mass 1.0f and shot from the height of 3.5f
Initial velocity provided to the projectile is vec3(8.0f, 7.0f, 5.0f)
Gravitational constant is assumed to be -9.8f
The simulation was captured for 10 secs with timestep equal to 0.1f

Observations:
RK4 integration has the highest accuracy, while the Euler method has the least.

The Verlet method allows simulating motion using only position, so it is less computationally expensive than other methods like RK4, and it doesn't suffer from energy drift as Euler does. But it can be less accurate.

Reflection:
Three methods of integration implemented with respect to simulation time including Runge Kutta. Application has the ability to switch between different methods at run-time via a GUI.

The integrators are written in a reusable way, to extend easily.

Criteria 1.3

Collision detection will be applied between cannon balls and everything in the game. Cannon balls will be fired by cannons and can also be collected and thrown by agents. In order to collide with the game, sphere-sphere, sphere-cylinder and sphere-cuboid collisions will need to be implemented.

Implementation details:

Naive implementation. Better implementation discussed in Criteria 1.5

Uses polymorphism to handle collisions between different types of objects in a reusable way defined in Projectile class.

```
void Projectile::HandleCollision(Projectile* other)
void Projectile::HandleCollision(Tower* other)
void Projectile::HandleCollision(Agent* other)
void Projectile::HandleCollision(Tile& other, vec3 n)
```

CASE I : Sphere to Sphere Collision
Considering different masses and radii, the collision detection is done via following condition

```
vec3 d = this->GetPosition() - other->GetPosition();
length(d) < other->GetBoundingRadius() + this->GetBoundingRadius()
```

CASE II : Sphere to General Cylinder Collision
Considering different radii for both objects, and given cylindrical object doesn't move after collision impact, the collision detection is done via following condition:

```
length(d) < GetBoundingRadius() + other->GetBaseRadius()
```

CASE III : Sphere to Plane/Cuboid Collision
Considering plane doesn't move after collision impact, the collision detection is done via following condition:

```
GetPosition().y + 0.5 <= 1.0f + GetBoundingRadius()
```

Note: 1.0f is representing the y=1.0f plane.

CASE IV: Sphere to Cuboid/Capsule/Compound Object Collision
The collision detection is done via following condition:

```
length(d) < GetBoundingRadius() + other->GetBaseRadius()
```

In the World's **Update( )** function, we check for any possible collisions between sphere and other entities i.e. with ground, wall, other spheres etc.

Code evidence and screenshots:

Filename: Game Entities/Projectile.h
Class names: Projectile
Method names: Method overloading - HandleCollision(CollidingObject*)

Reflection:
Can detect collisions between two spheres, and spheres and general cylinders – with different radii of colliding objects while conserving net energy with occasional energy dissipation.

Criteria 1.4

Collision response will occur when a cannon ball hits terrain and bounces off, or when a cannon ball hits another cannon ball.

Implementation details:
Collision response is done within the HandleCollision function defined in the Projectile class itself.

Whenever a ball hits a surface that does not move, an appropriate response is to reflect the ball's velocity with respect to the collision normal.

```
V = U - ( 1+e ) * ( N.U ) * N
```

To resolve the sphere to sphere collision with varying radii and masses,

We first create the normal vector

```
vec3 d = this->GetPosition() - other->GetPosition();
vec3 n = glm::normalize(d);
```

Calculate vL1 and vL2

```
vec3 vL1 =
     ((this->GetMass() - _RESTITUTION * other->GetMass()) *
     (dot(this->GetVelocity(), n) * n) +
     (this->GetMass() + _RESTITUTION * other->GetMass()) *
     (dot(other->GetVelocity(), n)) * n) /
     (this->GetMass() + other->GetMass());

vec3 vL2 =
     ((this->GetMass() + _RESTITUTION * other->GetMass()) *
     (dot(this->GetVelocity(), n)) * n) +
     (other->GetMass() - _RESTITUTION * this->GetMass()) *
     (dot(other->GetVelocity(), n)) * n /
     (this->GetMass() + other->GetMass());
```

Calculate resultant velocities

```
vec3 v1 = this->GetVelocity() - (dot(this->GetVelocity(), n) * n) +
vL1;
vec3 v2 = other->GetVelocity() - (dot(other->GetVelocity(), n) * n) +
vL2;
```

Update resultant velocities after impact

```
this->SetVelocity(v1);
other->SetVelocity(-v2);
```

The **conservation of energy** tells us that there should be the same amount of momentum immediately after the impact as there is at the time of impact.

Restitution (e) coefficient, maintained in **config** file, is 1 then there should be no energy loss, and the simulation should behave as it has previously. If e is set to 0 then objects will slide and/or stick together.

Code evidence and screenshots:

Filename: Game Entities/Projectile.h
Class names: Projectile
Method names: Method overloading - HandleCollision(CollidingObject*)

Reflection:
Inelastic collision with correct response taking into account mass of colliding objects, however the system is not very stable.

Criteria 1.5

A spatial segmentation method will be implemented to reduce the number of redundant calculations when performing collision detection. This solution could also be reused by agents to reduce their processing time.

Implementation details:
Data Structures: **BoundingVolumeHierarchy** and **BoundingSphere** class

Adding two new classes in Physics Engine, named **CollisionObject** and **CollisionDetector**.

The CollisionObject class would represent an individual object in the scene, with properties such as BoundingSphere or BoundingBox. All Physical entities can inherit CollisionObject class and override BoundingSphere method.

The BVH class would handle building and updating the hierarchical tree structure of the bounding volumes.

The CollisionDetector class would manage a list of all the objects in the scene and use the BVH to efficiently detect collisions between them. It would have methods for adding and removing objects, querying for collisions, and updating the BVH when objects move.

The CollisionDetector class can use the queryCollision method of BVH to check for collisions between the queried object and other objects. It can also maintain a vector of collision objects and add or remove them as necessary.

Overall, this design separates the responsibilities of building and traversing the BVH, representing individual objects, and managing the overall collision detection process, making it easy to understand and maintain.

Code evidence and screenshots:

Filename:
DataStructure/BoundingSphere.h
DataStructure/BoundingVolumeHierarchy.h
PhysicsEngine/CollisionObject.h
PhysicsEngine/CollisionDetector.h

Class names:
CollisionObject, CollisionDetector, BoundingSphere, BoundingVolumeHierarchy

Method names:
Merge, intersect, queryCollisions, create, insert, remove, update, split

Reflection:
Due to technical difficulties in implementation, the present solution doesn't include a working demo for this criteria.

Criteria 2.1

Action selection could be governed by a state machine and informed by fuzzy logic, that could consider the level of risk moving from one place to another according to proximity to enemy towers, the time it would take to move, the potential benefit in terms of power up boosts and the current state of the agent.

Implementation details:

Current implementation has limited states, however it can be easily extended to add states for implementing other strategies – pick up bonus items and defend.

Current State	Condition	State Transition
Spawn	Ready	Move
Move	At Tower; Opponent influenced	Attack
Move	At Tower; Neutral	Influence
Attack	Reduced 100% Opponent Health	Influence
Attack	Reduced 100% Self Health	Spawn
Influence	Reduced 100% Self Health	Spawn

Each of the possible states an **Agent** may access are derived from the **State** class, giving us these concrete classes:

Agent States:

MoveTowardsSpawnBase – In this state the agent will be respawned at the team spawn location. Once the agent is initialised, it changes it state to *MoveTowardsTower*

MoveTowardsTower – In this state the agent will walk towards a tower. If the agent reaches tower, it'll change state to *EnterIntoTower*

EnterIntoTower – In this state, if the tower is neutral, agent enters *InfluenceTower* state. If the tower is controlled by opponent team, the agent enters *CaptureTower* state.

InfluenceTower – In this state, if the agent influences the tower to score points. If the agent's health is reduced to zero, the agent enters *MoveToSpawnBase* state.

and **CaptureTower** – In this state, the agent attacks the agent controlling the tower. If the agent is able to eliminate the opponent, it then changes its state to *InfluenceTower*. If the agent's health is reduced to zero while attacking, the agent enters *MoveToSpawnBase* state.

The **Agent::m_pCurrentState** pointer is able to point to any of these states. When the Update method of Agent is called, it in turn calls the Execute method of the currently active state with the, this pointer as a parameter.

Each concrete state is implemented as a singleton object. This is to ensure that there is only one instance of each state, which agents share

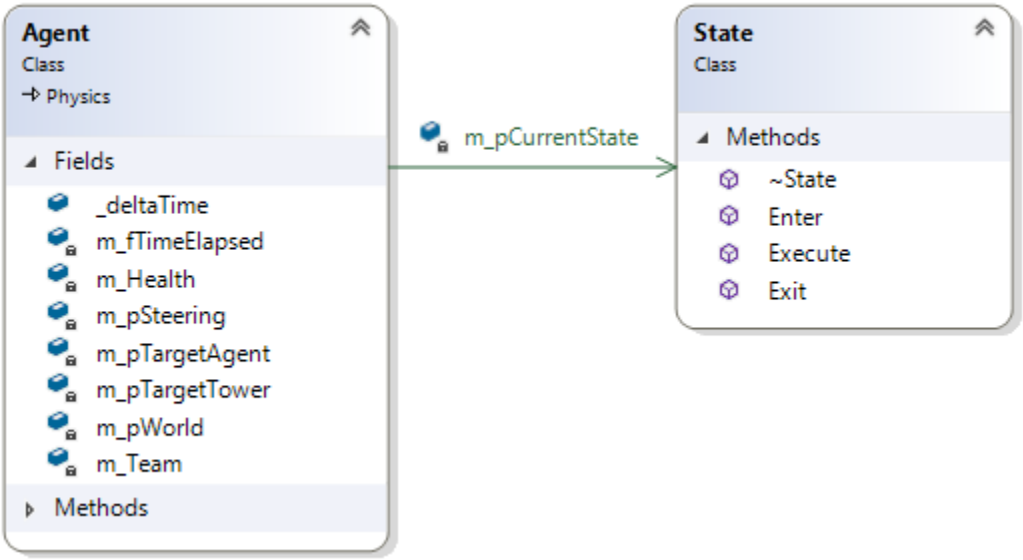
Code evidence and screenshots:

Filename:
GameEntities/Agent.h
AIModule/State.h
AIModule/StateFunctions.h

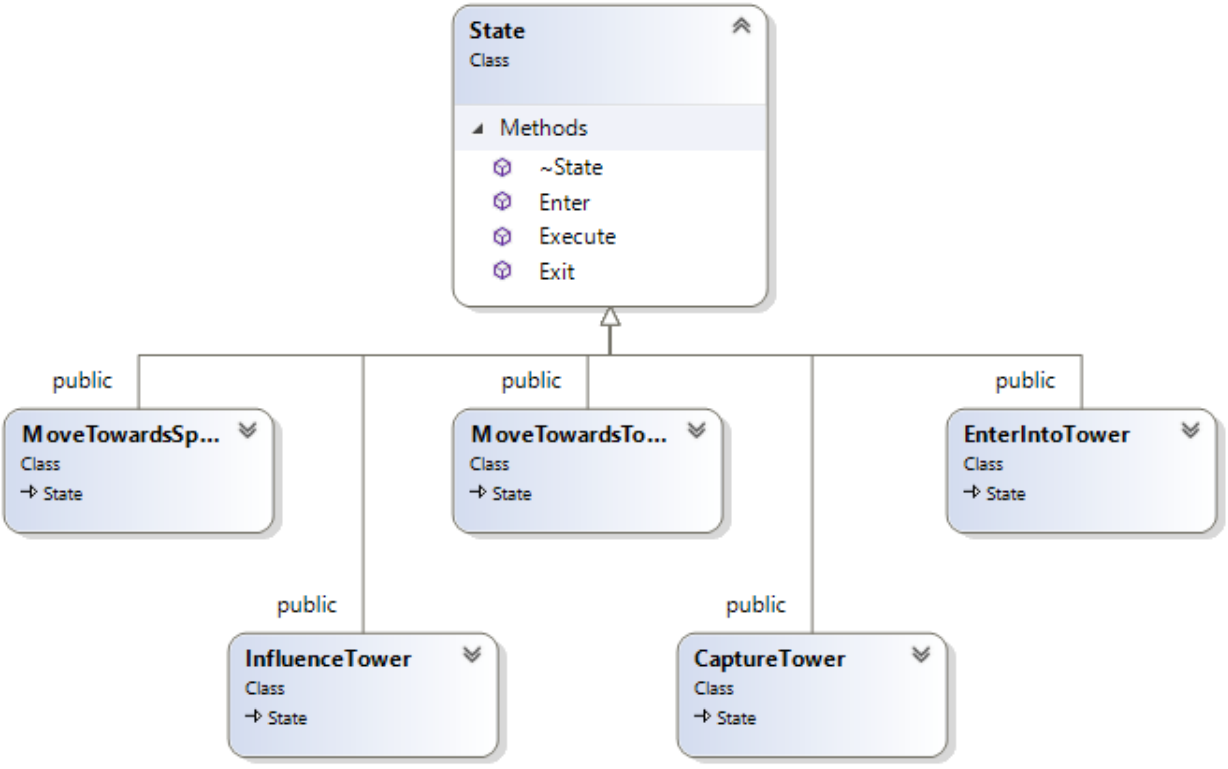
Class names:
Agent, State, MoveTowardsSpawnBase, MoveTowardsTower, EnterIntoTower, InfluenceTower, CaptureTower

Method names:
ChangeState, Enter, Execute, Exit

UML Diagram: Agent and State class.



UML Diagram: State and its concrete classes.



Reflection:

Agent states and transitions controlled by state machines.

Criteria 2.2

Demonstrating a variety of steering behaviours, combined according to some compound behaviour and could also include some group behaviours such as flocking or moving in formation.

Implementation details:

Seek – Moves the agent towards a target position
vec3 Seek(vec3 TargetPos);

Flee – Moves the agent away from a target position
vec3 Seek(vec3 TargetPos);

Arrive – Attempts to arrive at the target position with a zero velocity
vec3 Arrive(vec3 TargetPos, Deceleration deceleration);

Wander – Makes the agent wander about randomly
vec3 Wander();

Obstacle Avoidance – Returns a steering force which will attempt to keep the agent away from any obstacles it may encounter
vec3 ObstacleAvoidance(std::vector<GameEntity*>& obstacles);

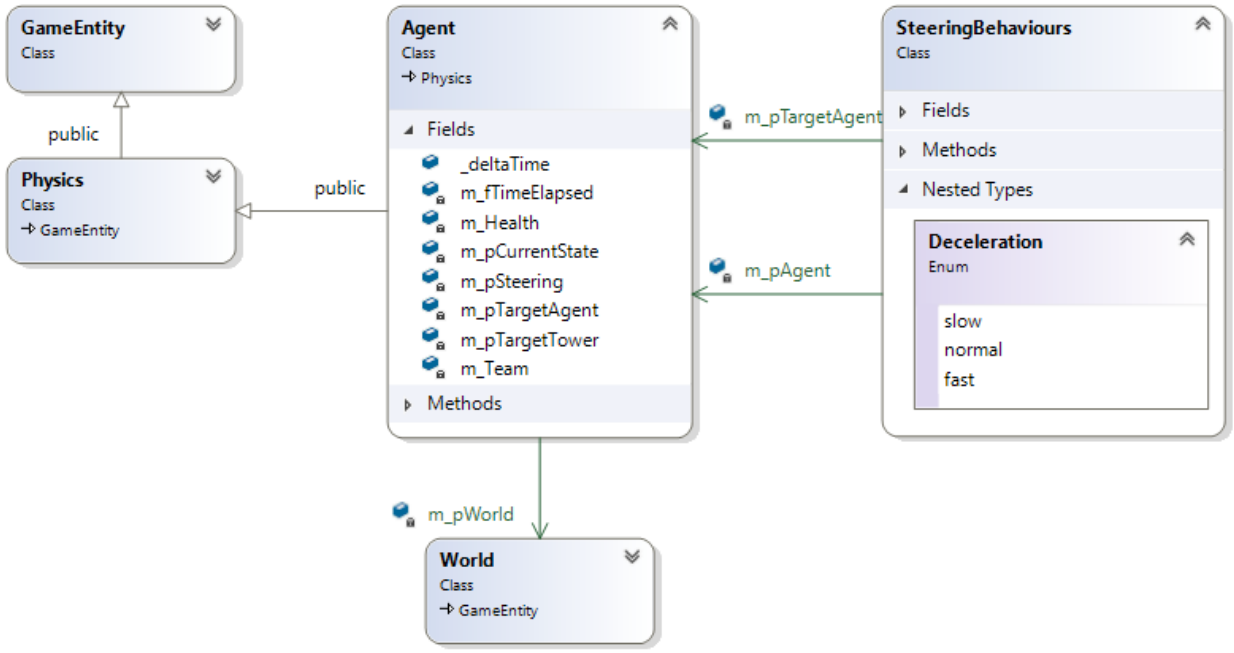
Wall Avoidance – Returns a steering force which will keep the agent away from any walls it may encounter
vec3 WallAvoidance(std::vector<GameEntity*>& walls);

Follow Path – Given a series of vec3 points, this method produces a force that will move the agent along the waypoints in order
vec3 FollowPath();

The **Calculate** function performs a weighted sum of all the moving behaviours and adding them together to create the resulting force. It resets the steering force and then adds each individual behaviour generated steering force to m_SteeringForce.

```
m_SteeringForce += Seek(pos) * m_WeightSeek;
m_SteeringForce += Flee(pos) * m_WeightFlee;
m_SteeringForce += Arrive(pos, m_Deceleration) * m_WeightArrive;
```

UML Diagram: Agent and its SteeringBehaviours classes.



Code evidence and screenshots:

Filename: AIModule/SteeringBehaviours.h
Class names: SteeringBehaviours
Method names: Calculate, Seek, Flee, Arrive, Wander, WallAvoidance, ObstacleAvoidance and FollowPath

Reflection:
Agents are able to move through the world using one Compound steering behaviour that combines simple steering behaviours.

Criteria 2.3

The terrain will be made up of cubes making a grid structure reasonably easy to create and perform pathfinding on. The paths could be weighted using length, or by adding a terrain type to enable faster or slower movement. The grid could also be replaced by a nav mesh.

Implementation details:

The data structure used in our implementation is a 2D array, which represents the map or graph on which the pathfinding is to be performed. Each cell in the array is represented by a Node class, which stores information about the cell such as its position, the cost of traversing it, and whether it is blocked or not. The A* algorithm uses this information to determine the best path from the starting point to the end point.

The A* algorithm uses a priority queue to store the nodes that need to be visited. The queue is sorted based on the estimated total cost of reaching the end point from each node. This cost is calculated using the Manhattan distance heuristic, which estimates the distance between two points based on the difference in their x and y coordinates.

The A* algorithm is used in the main game loop to calculate the path for the player character to move towards the destination.

Code evidence and screenshots:

Filename:
AIModule/Pathfinder.h
DataStructure/Node.h
Class names: Pathfinder, Node
Method names: addNeighborsToOpenList, calculateManhattanDistance, getNodeIndex

Reflection:
Path finding on grid attempted but unusable.

Criteria 2.4

The locomotion system will enable agents to actually move. Ideally this could be shared with physical objects.

Implementation details:

The init function is called to set up the locomotion system and associates the agent and the physics engine. The update function is called every frame and performs the necessary calculations to update the agent's position and velocity based on the current state of the system. The walk function is called to make the agent move in a certain direction, and the jump function is called to make the agent jump.

The class has a private member variable m_isJumping that keeps track of whether the agent is currently jumping or not. This variable is used in the update function to determine whether the agent should be affected by gravity or not.

The update function first checks if the agent is currently jumping. If it is, it applies an upward force to the agent, and checks if the agent has landed by comparing the ground height to the agent's current position. If the agent has landed, it sets the m_isJumping variable to false. If the agent is not jumping, the function applies gravity to the agent by applying a downward force. After that, it updates the agent's position and velocity using the physics engine.

The walk function applies a force to the agent in the desired direction, using the WALK_FORCE constant.

The jump function is called to make the agent jump, but it will only make the agent jump if it is not currently jumping. If the agent is jumping it does nothing.

The constants used in the functions such as JUMP_FORCE, WALK_FORCE, GRAVITY etc.. need to be adjusted according to the physics engine and the scenario of the simulation.

Code evidence and screenshots:

Filename: AIModule/LocomotionSystem.h
Class names: LocomotionSystem
Method names: Walk, Jump, Update

Reflection:
Attempted locomotion systems unusable..

Criteria 2.5

A spatial segmentation method should be implemented to reduce the number of redundant calculations when making decisions and simulating.

Implementation details: N/A

Code evidence and screenshots: N/A

Reflection:
Spatial segmentation technique not implemented.

Conclusion

Overall Evaluation and Reflection

Project requirements were clear and very well documented, which helped to design the solution with ease.

Scaffolding framework helped to take things off ground quickly.

The project presented a number of challenges, mostly in terms of practical approach for implementing real-time physics in simulation. The project implements most of the features and concepts learned during the trimester.

A number of features in the physics engine as well as intelligence module could be improved further.

Apart from technical aspects of the project, the gameplay could be enhanced further by implementing the remaining features.

References

Physics for Game Developers, 2nd Edition
Bryan Bywalec, David M Bourg
Publisher(s): O'Reilly Media, Inc.

Programming Game AI by Example
Mat Buckland
Wordware Publishing, Inc.