700110 ACW Report

Practical physical based model-ling and artificial intelligence

Criteria 1.1 Physical Simulation

We have to apply Newtonian physics in this section for the agents as well as for the cannon balls through the entire scene.

How was it achieved?

Newtonian physics has been implemented when agents move, and to cannon balls firing/thrown through the scene. *MovingEntity.h* class is a parent class inherited by *Ball.h* and *Agent.h*.

MovingEntity.h file contains member elements: Mass, Velocity, MaxSpeed, MaxForce, MaxTurnRate, ForwardHead, SideHead. All these elements are used by the two derived classes.

Ball.h file class includes some more forces such as Gravity, DragCoefficient and Thrust. To implement the projectile motion of the sphere all the forces are added up to find totalForce. Using this we calculate the acceleration. From this we get the newVelocity and newPosition of the sphere.

For the *Agent.h* file we use Seek steering behaviour to move our agent which uses agents MaxSpeed to get desired velocity. This desired velocity is the SteeringForce which we use to calculate the acceleration. Using this acceleration we calculate our newVelocity and then finally calculate our newPosition.

Where is it?

MovingEntity.h

```
class MovingEntity : public GameEntity
{
      private:
            float m_Mass;
            float m_MaxSpeed;
            float m_MaxForce;
            float m_MaxTurnRate;
            glm::vec3 m_ForwardHead;
            glm::vec3 m_SideHead;
            glm::vec3 m_Velocity;
      …
      …
}
```

Ball.h

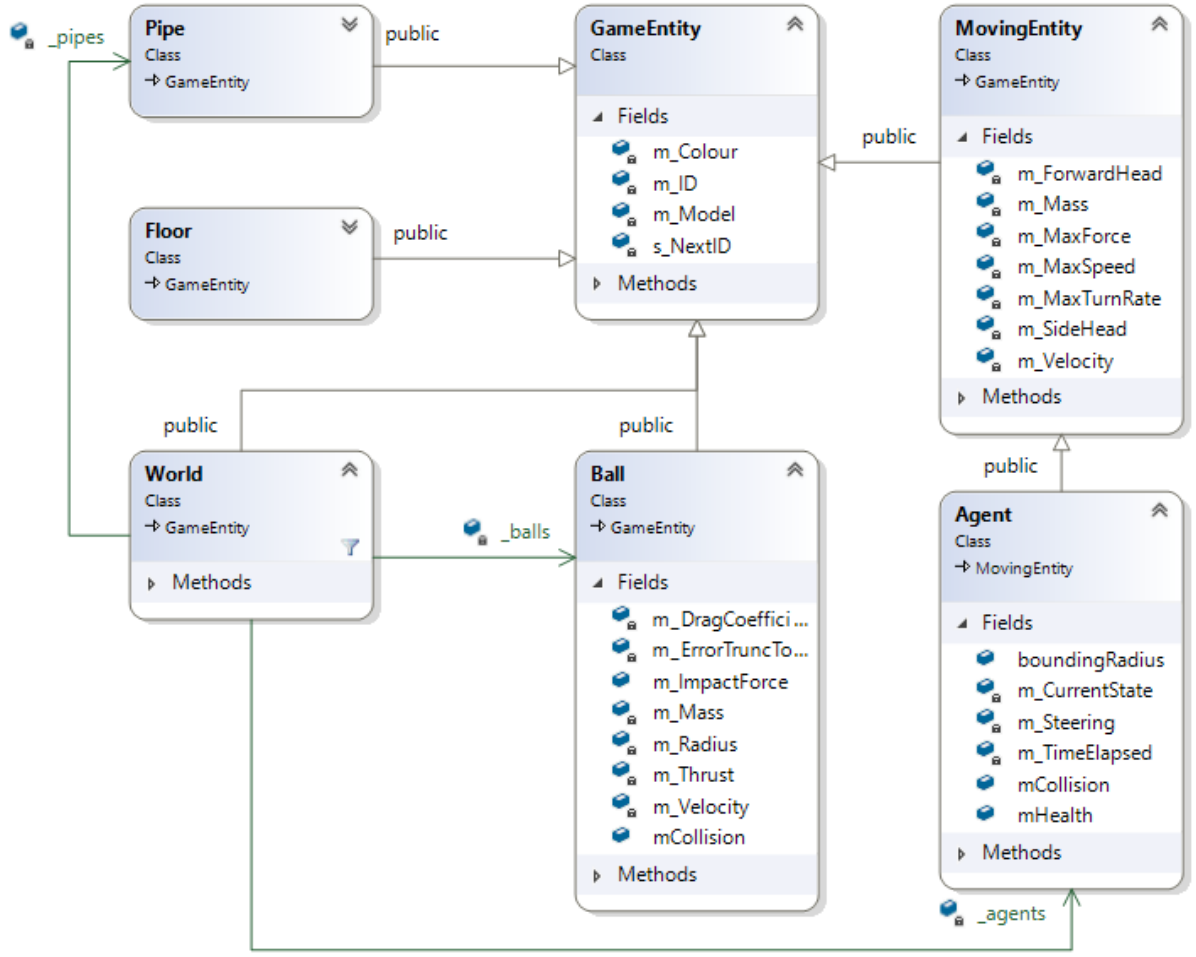
```
class Ball : public MovingEntity
{
private:

    …
    …
    glm::vec3 m_DragCoefficient;
    glm::vec3 m_Thrust;
    float m_ErrorTruncTolerance;
    …
    …
    void EulerBasicIntegration(float pDeltaTime)
   {
    …
      totalForce = gravity + m_Thrust - (m_DragCoefficient *
      m_Velocity);
    acceleration = totalForce / m_Mass;
    newVelocity = m_Velocity + acceleration * newDeltaTime;
    newPosition = GetPosition() + newVelocity * newDeltaTime;

    SetVelocity(newVelocity);
    SetPosition(newPosition);
   }
}
```

Agent.h

```
class Agent :  public MovingEntity
{
    …
    …
    Void update(float pDeltaTime)
    {
         float m_Mass = GetMass();
         glm::vec3 currentPosition = GetPosition();
         glm::vec3 currentVelocity = GetVelocity();

         glm::vec3 steeringForces = m_Steering->Seek(...);
         glm::vec3  acceleration = steeringForces / m_Mass;
```

```
            newVelocity = currentVelocity + acceleration * pDeltaTime;
            newPosition = currentPosition + newVelocity * pDeltaTime;

            SetPosition(newPosition);
            SetVelocity(newVelocity);
        }
}
```

Class Diagram



How well does it work?

All the Newtonian physics implemented in the code is working thoroughly without any errors. Newtonian physics has been applied accurately with pragmatic measures to prevent instability (i.e. capping velocities and forces) written in a reusable way.

Criteria 1.2 Numerical Integration

We have to apply numerical integration to all the moving objects and implement at least three methods of integration which should be possible to change at the run time.

Data should be collected from all three methods and comment should be made on the impact of the methods in terms of accuracy and performance

How was it achieved?

We have implemented three methods for numerical integration in our code namely, EulerBasicIntegration, ImprovedEulerIntegration, and RungeKutta4Integration with respect to simulation time.

We are using all of these integration methods in our *Ball.h* to make projectile motion for the sphere (canon balls). All the methods use Newtonian physics to calculate the projectile motion.

Finally in the update method of *World.h* we call these methods which can be switched during the run-time via GUI to simulate the projectile motion of sphere

Where is it?

Ball.h

```
class Ball : public MovingEntity
{
   …
   …
   void EulerBasicIntegration(float pDeltaTime) {
      glm::vec3 gravity(0.0f, -9.8f, 0.0f);
      glm::vec3 totalForce, acceleration, newVelocity, newPosition;

      float errorTruncation, newDeltaTime;
      glm::vec3 v1, v2;

      totalForce = gravity + m_Thrust - (m_DragCoefficient *
      m_Velocity);
      acceleration = totalForce / m_Mass;
      v1 = m_Velocity + acceleration * pDeltaTime;

      totalForce = gravity + m_Thrust - (m_DragCoefficient *
      m_Velocity);
      acceleration = totalForce / m_Mass;
```

```
    v2 = m_Velocity + acceleration * (pDeltaTime / 2);

    totalForce = gravity + m_Thrust - (m_DragCoefficient * v2);
    acceleration = totalForce / m_Mass;
    v2 = v2 + acceleration * (pDeltaTime / 2);

    errorTruncation = glm::abs(glm::length(v1) - glm::length(v2));
    newDeltaTime = pDeltaTime * glm::sqrt(m_ErrorTruncTolerance /
    errorTruncation);

    if (newDeltaTime < pDeltaTime)
    {
    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    m_Velocity);
    acceleration = totalForce / m_Mass;
    newVelocity = m_Velocity + acceleration * newDeltaTime;
    newPosition = GetPosition() + newVelocity * newDeltaTime;
    }
    else
    {
    newVelocity = v1;
    newPosition = GetPosition() + newVelocity * pDeltaTime;
    }

    SetVelocity(newVelocity);
    SetPosition(newPosition);
}

void ImprovedEulerIntegration(float pDeltaTime)
{
    glm::vec3 gravity(0.0f, -9.8f, 0.0f);
    glm::vec3 totalForce, acceleration, newVelocity, newPosition;
    glm::vec3 k1, k2;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    m_Velocity);
    acceleration = totalForce / m_Mass;
    k1 = acceleration * pDeltaTime;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    (m_Velocity + k1));
```

```
    acceleration = totalForce / m_Mass;
    k2 = acceleration * pDeltaTime;

    newVelocity = m_Velocity + (k1 + k2) / 2.0f;
    newPosition = GetPosition() + newVelocity * pDeltaTime;

    SetVelocity(newVelocity);
    SetPosition(newPosition);
}

void RungeKutta4Integration(float pDeltaTime)
{
    glm::vec3 gravity(0.0f, -9.8f, 0.0f);
    glm::vec3 totalForce, acceleration, newVelocity, newPosition;
    glm::vec3 k1, k2, k3, k4;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    m_Velocity);
    acceleration = totalForce / m_Mass;
    k1 = acceleration * pDeltaTime;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    (m_Velocity + k1 / 2.0f));
    acceleration = totalForce / m_Mass;
    k2 = acceleration * pDeltaTime;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    (m_Velocity + k2 / 2.0f));
    acceleration = totalForce / m_Mass;
    k3 = acceleration * pDeltaTime;

    totalForce = gravity + m_Thrust - (m_DragCoefficient *
    (m_Velocity + k3));
    acceleration = totalForce / m_Mass;
    k4 = acceleration * pDeltaTime;

    newVelocity = m_Velocity + (k1 + 2.0f * k2 + 2.0f * k3 + k4) /
    6.0f;
    newPosition = GetPosition() + newVelocity * pDeltaTime;

    SetVelocity(newVelocity);
```

```
      SetPosition(newPosition);
   }
}
```

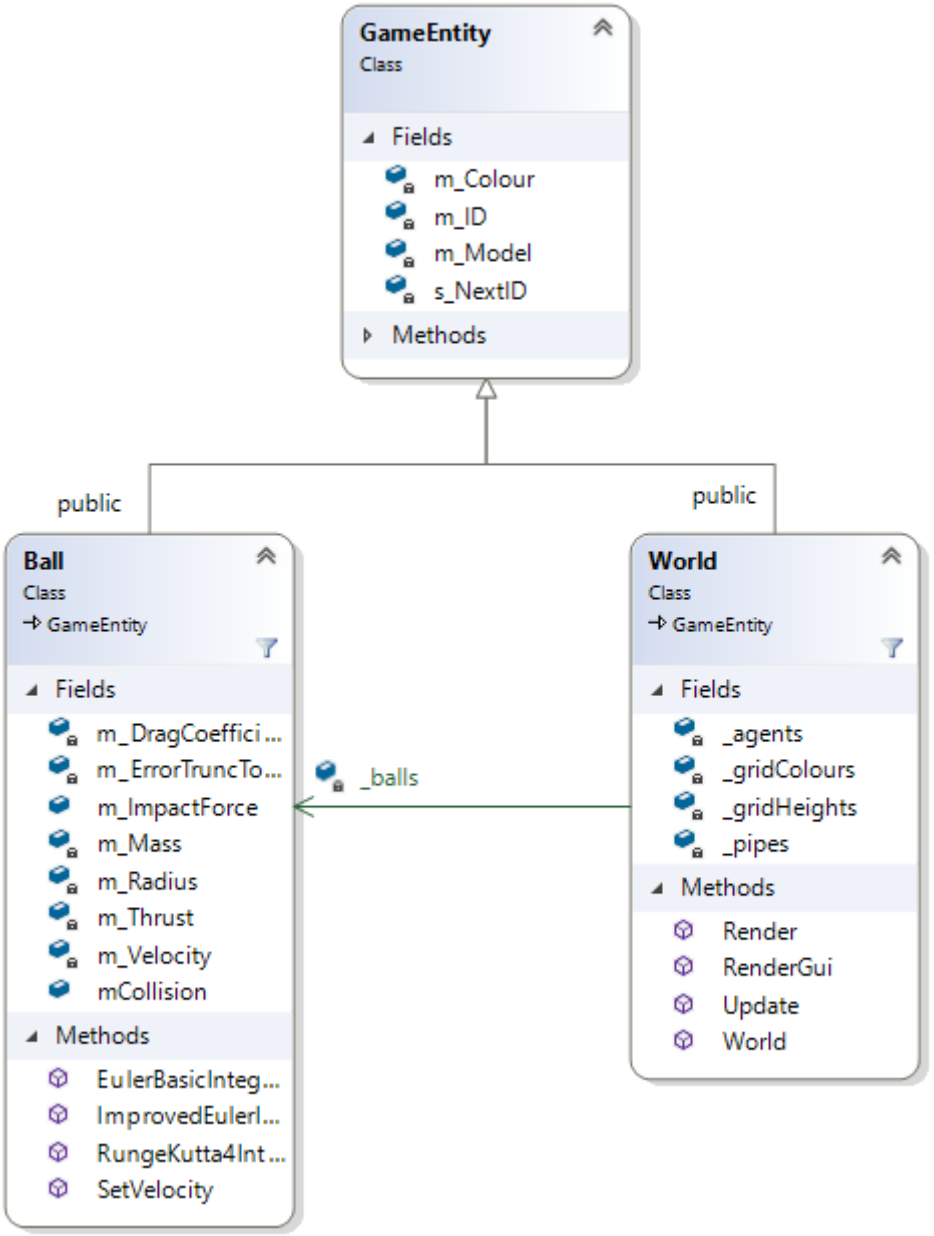
World.h

```
class World :public GameEntity
{
   …
   …
   void Update(float pSeconds)
   {
      …
      …
      for (int i = 0; i < NUMBER_OF_BALLS; i++)
      {
            if (GetIntegrationMethod() == 0)
           {
                _balls[i].EulerBasicIntegration(pSeconds);
           }
           else if (GetIntegrationMethod() == 1)
           {
                _balls[i].ImprovedEulerIntegration(pSeconds);

           }
           else if (GetIntegrationMethod() == 2)
           {
                _balls[i].RungeKutta4Integration(pSeconds);
           }
      }
   }
   …
   …
}
```

App.cpp

```
void renderImGui()
{
    ImGui::BeginGroup();
    ImGui::Text("Integration Method");
    auto integration = world.GetIntegrationMethod();
     ImGui::Combo("", &integration, "Euler\0Improved
     Euler\0RK4\0\0");
    ImGui::EndGroup();

    if (integration == 0) {
        world.SetIntegrationMethod(0);
    }
    else if (integration == 1) {
        world.SetIntegrationMethod(1);
    }
    else if (integration == 2) {
        world.SetIntegrationMethod(2);
    }

    ImGui::End();
}
```

Class Diagram



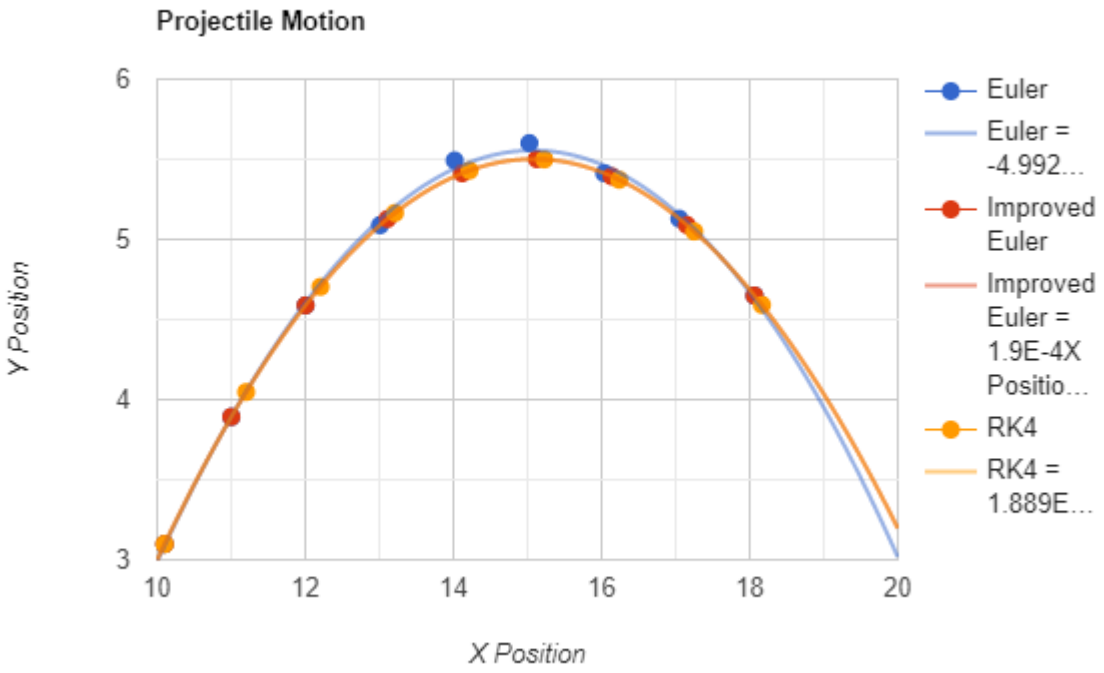
How well does it work?

All the three methods of integration are working which can also be switched during run time using GUI.

Some indication of the impact of the various methods:
Data has been collected for position of sphere over a specific period of time while in projectile motion for all the three methods (where Z is constant):

EulerBasicIntegration		ImprovedEulerIntegration		RungeKutta4Integration	
X	Y	X	Y	X	Y
10.1	3.09804	10.1	3.09804	10.1	3.09804
11.0011	3.8922	11.0011	3.8922	11.2016	4.04712
12.0042	4.5884	12.0042	4.5884	12.2051	4.70412
13.0093	5.0886	13.1099	5.12784	13.2106	5.16512
14.0164	5.4928	14.1172	5.41244	14.2181	5.43012
15.0255	5.6011	15.1265	5.50104	15.2276	5.49912
16.0366	5.4132	16.1378	5.39364	16.2391	5.37212
17.0497	5.1294	17.1511	5.09024	17.2526	5.04912
18.0648	4.6496	18.0648	4.6496	18.1664	4.59084

Using these points the graph can be plotted as follows:



We can see from the graph that the RK4 method is giving more realistic projectiles when compared to the other two methods.

Criteria 1.3 Collision Detection

We have to show collision detection in this section between cannon balls and everything in the game. We have to show sphere-sphere, sphere-cylinder and sphere-cuboid collisions.

How was it achieved?

Collision detection methods have been implemented inside of the *World.h* file for collision detection of cannon balls with the ground, with walls of the boundary, with other balls as well as with towers.

Collision detection between sphere and ground is done by declaring a global variable _GROUND_PLANE (which is set to be 1 as our floor is y = 1) which we add with the radius of the sphere. If this is less than Y position of the sphere then a collision is detected.

Collision detection between two spheres is done by calculating the distance between the centres of the two spheres. If the distance is less than or equal to the sum of the two sphere's radii, then a collision has occurred.

Collision detection between a sphere and a cylinder is done by first determining the distance between the sphere's centre and the cylinder's axis is less than the sum of the sphere's radius and cylinder's radius.

The boundary walls of the map have been set to a fixed value according to their position. If the radius of the sphere is less than these fixed values then a collision is detected between sphere and walls.

Where is it?

World.h

```
class World :public GameEntity
{
	…
	…
	void SphereGroundCollision(Ball* b)
	{
		…
		…
		if (b->GetPosition().y  <= (_GROUND_PLANE +
		b->GetRadius()))
		{
			//code for collision response
		}
		…
```

```
}

bool BallWallCollisions(Ball* b)
{
      float RWlength, LWlength, FWlength, BWlength;
      RWlength = (GRID_WIDTH - 1.5) - b->GetPosition().x;
      LWlength = b->GetPosition().x - 0.5;
      FWlength = (GRID_DEPTH - 1.5) - b->GetPosition().z;
      BWlength = b->GetPosition().z - 0.5;

      if (RWlength < b->GetRadius() || LWlength <
      b->GetRadius() || FWlength < b->GetRadius() || BWlength <
      b->GetRadius())
      {
            //code for collision response
      }
     return true;

}

void SphereSphereCollision(Ball* b1, Ball* b2)
{
      …
      …
      if (glm::length(d) < b2->GetRadius() + b1->GetRadius())
      {
            //code for collision response
      }
}

bool SphereCylinderCollision(Ball* b)
{
      for (int i = 0; i < NUMBER_OF_PIPES; i++)
       {
            …

            // Vector from sphere centre to cylinder centre

            glm::vec3 d = b->GetPosition() -
            _pipes[i].GetPosition();
```

```
                // Project d onto cylinder axis

                float dist = glm::dot(d, glm::vec3(0, 1, 0));

                // Check if sphere is within cylinder height
                float Height = _pipes[i].m_BaseHeight +
                _pipes[i].m_LipHeight;

                if (dist < 0 || dist > Height)
                {
                     return false;
                }

                // Check if distance is less than sphere radius plus
                cylinder radius
                if (glm::length(d) < b->GetRadius() +
                _pipes[i].m_BaseRadius)
                {
                     //code for collision response
                     return true;
                }
           }
           return false;
     }
     …
     …
}
```

How well does it work?

The methods have been written in a reusable way such that we can detect collisions between sphere-sphere and sphere-cylinder of varying radii as well as sphere-ground and sphere-walls.

Criteria 1.4 Collision Response

We have to show collision response in this section for a cannon ball when it hits another cannon ball with different mass and size as well as when the cannon ball hits the terrain.

How was it achieved?

Collision detection methods implemented in the previous section are further expanded to calculate a few extra variables used to set new velocity of the sphere as part of collision response if a collision has been detected.

For sphere ground collision response we use the formula $V = U - 1 + e * N.U * N$ to calculate the new velocity of sphere where U is the current sphere velocity, e is coefficient of restitution and N is the ground normal.

For sphere sphere collision response we use the following formula to calculate velocities of two spheres after the collision.
$\mathbf{v}N1 = (m1 - em2)\ (\mathbf{u}1.\mathbf{N})\ \mathbf{N} + (m2 + em2)\ (\mathbf{u}2.\mathbf{N})\ \mathbf{N} / (m1 + m2)$
$\mathbf{v}N2 = (m1 + em1)\ (\mathbf{u}1.\mathbf{N})\ \mathbf{N} + (m2 - em1)\ (\mathbf{u}2.\mathbf{N}\ )\mathbf{N} / (m1 + m2)$
v1 = u1 − u1.N N+vN1
v2 = u2 − u2.N N+vN2
Ui are the velocities of the particles before impact
Vi are the velocities of the particles after impact
N is the line connecting the centres of the particles (normalised)

For sphere cylinder collision response we first calculate collision normal which is a vector that points away from the surface of the cylinder and towards the centre of the sphere. It can be calculated by normalising the vector pointing from the sphere's centre to the closest point on the cylinder's axis. After this the sphere's new velocity can be calculated using the glm::reflect function, which reflects a vector about a given normal. The sphere's velocity is reflected by the collision normal.

Collision response for the sphere wall is an elastic collision where no kinetic energy is lost and the sphere is rebounded back from the wall with the same velocity but in opposite direction.

Where is it?

World.h

```
class World : public GameEntity
{
      …
      …
      void SphereGroundCollision(Ball* b)
      {
```

```
      float e = 0.6f;
      glm::vec3 U = b->GetVelocity();
      glm::vec3 GroundNormal = {0,1,0};
      glm::vec3 V = U - ((1.0f + e) * ((glm::dot(GroundNormal,
      U)) * GroundNormal));

      if(collision detected)
      {
            b->SetVelocity(glm::vec3(V.x , V.y , V.z));
      }
}

bool BallWallCollisions(Ball* b)
{
      …
      …
      if(collision detected)
      {
            b->SetVelocity(-(b->GetVelocity()));
      }
      …
}

void SphereSphereCollision(Ball* b1, Ball* b2)
{
      float e = 1.0f;
      glm::vec3 d = b1->GetPosition() - b2->GetPosition();
      glm::vec3 N = glm::normalize(d);

      glm::vec3 VN1 = (((b1->GetMass() - (e * b2->GetMass())) *
      (glm::dot(b1->GetVelocity(), N)) * N) + ((b2->GetMass() +
      e * b2->GetMass()) * (glm::dot(b2->GetVelocity(), N)) *
      N)) / (b1->GetMass() + b2->GetMass());

      glm::vec3 VN2 = (((b1->GetMass() + (e * b1->GetMass())) *
      (glm::dot(b1->GetVelocity(), N)) * N) + ((b2->GetMass() -
      e * b1->GetMass()) * (glm::dot(b2->GetVelocity(), N)) *
      N)) / (b1->GetMass() + b2->GetMass());

      glm::vec3 V1 = b1->GetVelocity() -
      (glm::dot(b1->GetVelocity(), N) * N) + VN1;
```

```
            glm::vec3 V2 = b2->GetVelocity() -
            (glm::dot(b2->GetVelocity(), N) * N) + VN1;

            if(collision detected)
            {
                  b1->SetVelocity(V1);
                  b2->SetVelocity(-V2);
            }
      }

      bool SphereCylinderCollision(Ball* b)
      {
            for (int i = 0; i < NUMBER_OF_PIPES; i++)
            {
                  glm::vec3 U = b->GetVelocity();
                  glm::vec3 d = b->GetPosition() -
                  _pipes[i].GetPosition();
                  glm::vec3 N = glm::normalize(d);
                  glm::vec3 V = glm::reflect(b->GetVelocity(), N);
                  …
                  …
                  if(collision detected)
                  {
                        b->SetVelocity(V);
                  }
            }
      }
      …
}
```

How well does it work?

Sphere-sphere and Sphere-Ground collision are inelastic collisions with correct response taking into account mass of colliding objects. Sphere-cylinder and Sphere-wall collisions are elastic collisions with correct response taking into account mass of colliding objects.

Criteria 1.5 Spatial Segmentation

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.

How was it achieved?

The basic idea behind spatial segmentation is to only check for collisions between objects that are in the same cell or nearby cells, rather than checking for collisions between all objects in the entire environment. This can significantly reduce the number of redundant calculations and improve performance.

Not implemented

Where is it?

Not implemented

How well does it work?

Not implemented

Criteria 2.1 Action Selection

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.

How was it achieved?

Action selection implemented in the code is simple rule-based decision making without an explicit state machine. The actions for agents are move, influence, hit.

Agents initial action is to Seek() towards a tower closer to them. In *Agent.cpp* we have defined a method RedTeam1Update() that will move the agent towards the tower nearest to it.

If the distance between agent and tower's base radius is less than the tower's safe zone then agent will influence the tower to change colour according to its team. In the *World.h* file we have defined a method ChangeTowerColor() for this action.

When the distance between agent and centre of a cannon ball is less than the radius of the cannon ball then it will call for 'Agent HIT' action and the health of the agent will decrease.In the *World.h* file we have defined a method AgentHit() for this action.

Where is it?
Where is this achievement evidenced in the code?

Agent.cpp

```
void Agent::RedTeam1Update(float pDeltaTime)
{
        …
        …
        glm::vec3 steeringForces = m_Steering->Seek(nearestTowerLocation);
        …
}
```

We call this Update method for RedTeam1 in the Update of World.h

World.h

```
class World :public GameEntity
{
	void ChangeTowerColor(Agent* a)
	{
		float d = glm::length(a->GetPosition() -
		nearestTowerLocation);
		float d1 = glm::length(a->GetPosition() -
		nearestTowerLocation);
		float d2 = glm::length(a->GetPosition() -
		nearestTowerLocation);
		float d3 = glm::length(a->GetPosition() -
		nearestTowerLocation);
		if (d < 2.0) {
			r1_Influence += 1;
		}
		else if (d1 < 2.0) {
			b1_Influence += 1;
		}
		else if (d2 < 2.0) {
			r2_Influence += 1;
		}
		else if (d3 < 2.0) {
			b2_Influence += 1;
		}

		if (r1_Influence == 10)
		{
			_pipes[0].SetColour(glm::vec3(1, 0, 0));
		}
		else if (b1_Influence == 10)
		{
		   _pipes[1].SetColour(glm::vec3(0, 0, 1));
		}
		else if (r2_Influence == 10)
		{
		   _pipes[2].SetColour(glm::vec3(1, 0, 0));
		}
		else if (b2_Influence == 10)
		{
		   _pipes[3].SetColour(glm::vec3(0, 0, 1));
```

```
            }
      }

      void AgentHit(Agent* a, Ball* b)
      {
            glm::vec3 D = a->GetPosition() - b->GetPosition();
            float length = glm::length(D);
            if (length < b->GetRadius() + a->boundingRadius)
            {
                  std::cout << "Agent HIT!" << std::endl;
                  a->mHealth -= 1;
            }
      }

      void Update(float pSeconds)
      {
            …
            …
            _agents[0].RedTeam1Update(pSeconds);
      }
}
```

How well does it work?

Some decision making rules are based on hard coded values which works well for this simulation but if the game entities increase it will be difficult to scale as explicit state machines have not been made.

Criteria 2.2 Steering Behaviours

We have to show a variety of steering behaviours and include a summary of each of the behaviours and what they do.
The steering behaviours will be combined according to some compound behaviour and could also include some group behaviours such as flocking or moving in formation

How was it achieved?
Extent of the achievement and how it was achieved.

SteeringBehaviour.h file contains definitions of different steering behaviours such as seek, arrive etc.

Seek() behaviour takes in a target position as input and then calculates desired velocity by multiplying the agent's max speed and normalized distance between agent's position and the target position. It then subtracts the agent's current velocity with the desired velocity and returns that value. In Agent's update method Seek() is called and its value is stored as the steeringForce which is used to calculate the new velocity of the agent and finally calculate the new position of agent.

Seek() is useful for getting an agent moving in the right direction, but often you'll want your agents to come to a gentle halt at the target position. For this reason one more steering behaviour Arrive() has been defined in SteeringBehaviour.h file. Similar to Seek(), it takes target position as input but it also takes a second input in the form of an enum 'Deceleration' which is used to decide how fast or slow the agent should arrive at the target position. Now because Deceleration is enumerated as an int, a DecelerationTweaker is required to provide fine tweaking of the deceleration. The method then calculates the desired velocity and subtract it from the agent's current velocity and returns it. In the update method of agent Arrive() is called and its value is stored as steeringForce which is then used to calculate the new velocity and position of the agent.

Where is it?

SteeringBehaviour.h

```
class SteeringBehavior
{
	private:
		glm::vec3 m_SteeringForce;

	public:
		glm::vec3 GetSteeringForce() const
		{
			return m_SteeringForce;
		}
		enum Deceleration{slow = 3, normal = 2, fast = 1 };
```

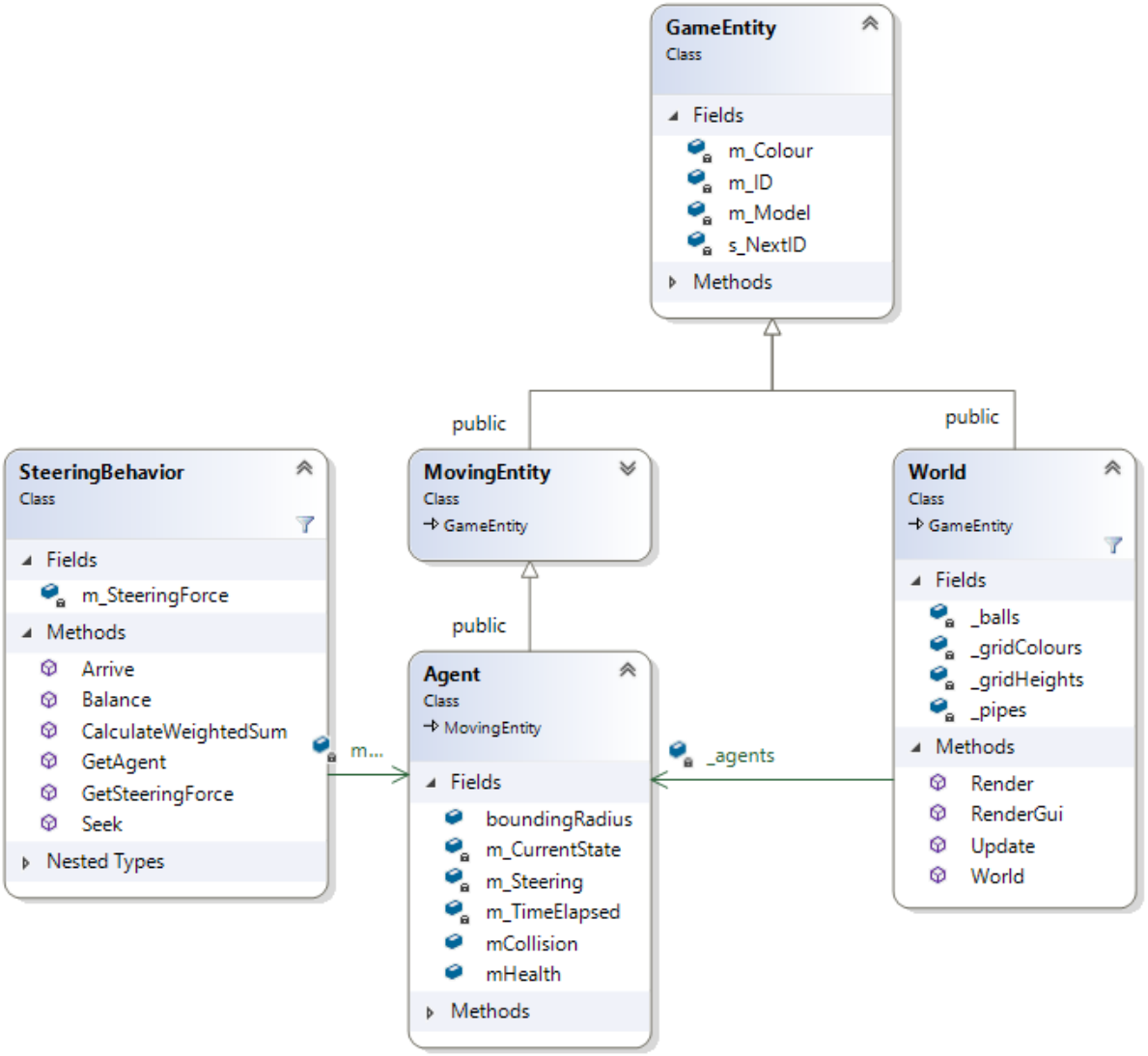
```
glm::vec3 CalculateWeightedSum()
{
     m_SteeringForce = glm::vec3(1.0f, 0.0f, 0.5f);
         return m_SteeringForce;
}

glm::vec3 Balance()
{
         m_SteeringForce = glm::vec3(0.5f, 0.0f, 0.5f);
         return m_SteeringForce;

}

glm::vec3 Seek(glm::vec3)
{
     glm::vec3 desiredVelocity = glm::normalize(pTarget -
     m_Agent->GetPosition()) * m_Agent->GetMaxSpeed();

     return desiredVelocity - m_Agent->GetVelocity();

}

glm::vec3 Arrive(glm::vec3, Deceleration)
{
     glm::vec3 ToTarget = pTarget -
     m_Agent->GetPosition();

    //Calculate the distance to the target position
    double dist = glm::length(ToTarget);

    if(dist > 0)
    {
     //because deceleration is enumerated as an int, this
     value is required to provide fine tweaking of the
     deceleration.
        const double DecelerationTweaker = 0.1;

     //Calculate the speed required to reach the target
     given the desired deceleration
```

```
                double speed = dist / ((double)deceleration *
                DecelerationTweaker);

                   //make sure the velocity does not exceed the max
                speed = glm::min(static_cast<float>(speed),
                m_Agent->GetMaxSpeed());

                glm::vec3 DesiredVelocity = ToTarget *
                static_cast<float>(speed / dist);

                return (DesiredVelocity - m_Agent->GetVelocity());

            }
            return glm::vec3(0);
        }
};
```

Agent.cpp

```
void Agent::Update(float pDeltaTime)
{
     …
     …
     glm::vec3 steeringForce = m_Steering->Seek(targetPosition);
     glm::vec3 steeringForce1 = m_Steering->Arrive(targetPosition,
     SteeringBehavior::Deceleration(1));

     glm::vec3 newVelocity, newPosition;

     newVelocity = currentVelocity + steeringForces * pDeltaTime;
     newPosition = currentPosition + newVelocity * pDeltaTime;

     SetPosition(newPosition);
     SetVelocity(newVelocity);

}
```

Class Diagram



How well does it work?

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

Criteria 2.3 Pathfinding

We have to perform path finding in this section and include a summary of how the path finding is achieved, a description of the data structure and algorithms implemented.
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.

How was it achieved?

Pathfinding has been implemented for agents on a terrain using the A* algorithm but unusable. The terrain is represented as a 2D grid of cells, where each cell has properties such as x and y coordinates, g, h, f values and walkability. The open list and closed list are used to keep track of the cells that have been searched and the path is retraced from the end cell to the start cell. The heuristic function used here is Manhattan distance.

Where is it?

PathFinding.h

```
class Pathfinding
{
  Public:
           // Define the terrain dimensions and initialise the
           terrain cells
              int width, height;
              std::vector<std::vector<Cell>> terrain;

           // Initialize the start and end positions
               int startX, startY, endX, endY;

           // Initialize the open and closed lists
                 std::priority_queue<Cell, std::vector<Cell>,
                 std::greater<Cell>> openList;
                std::unordered_map<int, int> closedList;

           // Initialize the path
               std::vector<Cell> path;

           // Constructor
           Pathfinding(int width, int height, int startX, int
           startY, int endX, int endY) {
            this->width = width;
```

```
        this->height = height;
        this->startX = startX;
        this->startY = startY;
        this->endX = endX;
        this->endY = endY;

        // Initialize the terrain cells
        for (int i = 0; i < width; i++) {
            std::vector<Cell> row;
            for (int j = 0; j < height; j++) {
                Cell cell;
                cell.x = i;
                cell.y = j;
                  cell.walkable = true; // assume all cells are
                  walkable
                row.push_back(cell);
            }
            terrain.push_back(row);
        }
    }

    // Find the path
    bool findPath()
{
      // Set the start cell
     terrain[startX][startY].g = 0;
     terrain[startX][startY].h = heuristic(startX, startY);
      terrain[startX][startY].f = terrain[startX][startY].g +
      terrain[startX][startY].h;
     terrain[startX][startY].parent = nullptr;
     openList.push(terrain[startX][startY]);

     // Keep searching while the open list is not empty
     while (!openList.empty()) {
            // Get the cell with the lowest f value from the
            open list
         Cell current = openList.top();
         openList.pop();

         // Add the current cell to the closed list
         closedList[current.x * width + current.y] = 1;
```

```
// Check if we have reached the end cell
if (current.x == endX && current.y == endY) {
    // We have found the path, so retrace it
    retracePath(current);

    return true;
}

// Check the neighbouring cells
for (int i = -1; i <= 1; i++) {
    for (int j = -1; j <= 1; j++) {
        // Skip the current cell
        if (i == 0 && j == 0) {
            continue;
        }

        // Get the x and y coordinates of the
   neighbouring cell
        int x = current.x + i;
        int y = current.y + j;

        // Check if the neighbouring cell is within
   the terrain bounds
        if (x < 0 || x >= width || y < 0 || y >=
   height) {
            continue;
        }

        // Check if the neighbouring cell is walkable
        if (!terrain[x][y].walkable) {
            continue;
        }

        // Check if the neighbouring cell is already
        in the closed list
        if (closedList.find(x * width + y) !=
   closedList.end()) {
            continue;
        }
```

```
                // Calculate the new g value for the
            neighbouring cell
                int newG = current.g + 1;

                // Check if the neighbouring cell is already
            in the open list
                if (openList.find(terrain[x][y]) !=
            openList.end()) {
                // Update the neighbouring cell if the new g
            value is lower
                    if (newG < terrain[x][y].g) {
                        terrain[x][y].g = newG;
                        terrain[x][y].h = heuristic(x, y);
                           terrain[x][y].f = terrain[x][y].g +
                      terrain[x][y].h;

                      terrain[x][y].parent =
                      &terrain[current.x][current.y];
                    }
                }
                else {
                      // Add the neighbouring cell to the open
                list
                    terrain[x][y].g = newG;
                    terrain[x][y].h = heuristic(x, y);
                      terrain[x][y].f = terrain[x][y].g +
                terrain[x][y].h;
                      terrain[x][y].parent =
                &terrain[current.x][current.y];
                    openList.push(terrain[x][y]);
                }
             }
          }
     }
     // We have not found a path
       return false;
}

   // Heuristic function
   int heuristic(int x, int y)
{
```

```
            // Manhattan distance
            return abs(x - endX) + abs(y - endY);

        }

        // Retrace the path
        void retracePath(Cell end)
        {
            // Start at the end cell
            Cell current = end;

            // Go through the cells in reverse order
            while (current.parent != nullptr) {
                // Add the cell to the path
                path.push_back(current);

                // Move to the parent cell
                current = *current.parent;

                // Reverse the path
                std::reverse(path.begin(), path.end());
            }
        }


};
```

How well does it work?

Path finding on the grid has been attempted but due to technical errors it is unusable.

Criteria 2.4 Locomotion Systems

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

How was it achieved?

The locomotion system for our agents is based on the steering behaviours that we have defined in our SteeringBehaviour.h file.

We use Seek() and Arrive() steering behaviour to actually move the agents. The locomotion system takes into consideration the physics of the game. It uses the mass, velocity, maxspeed attribute of agents to calculate the steeringforce which is then used to calculate the new velocity and position of the agent.

Where is it?

SteeringBehaviour.h

```
class SteeringBehavior
{
	…
	…
	…
	glm::vec3 Seek(glm::vec3)
	{
		glm::vec3 desiredVelocity = glm::normalize(pTarget -
		m_Agent->GetPosition()) * m_Agent->GetMaxSpeed();

		return desiredVelocity - m_Agent->GetVelocity();
	}

	glm::vec3 Arrive(glm::vec3, Deceleration)
	{
		…
		…
		//make sure the velocity does not exceed the max
		speed = glm::min(static_cast<float>(speed),
		m_Agent->GetMaxSpeed());
		…
		…
		return (DesiredVelocity - m_Agent->GetVelocity());
	}
}
```

How well does it work?

Locomotion system for the game has been integrated with physics and written in a reusable way

Criteria 2.5 Spatial Segmentation

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

How was it achieved?
Not implemented

Where is it?
Not implemented

How well does it work?
Not implemented

Evaluation

Newtonian physics is applied accurately along with different numerical integration methods. Collisions are detected between different game entities (with varying radii). Collision response gives correct response taking into account mass of colliding objects. Some methods for it are inelastic collision (Sphere-ground, Sphere-Sphere) but some have elastic collision as well (Sphere-wall).

Agent states are not defined and the agent only follows a fixed behaviour. Steering Behaviours have been implemented in the code. The locomotion system takes into consideration the physics of the game. Spatial segmentation is not implemented.

Following improvements can be done in the future:

- Agent states and transitions could be controlled by state machines, with decisions driven using fuzzy logic rather than crisp logic.
- Including group behaviours (like flocking, moving in formation) for the agents.
- Agents able to find a path on weighted graph-based representation of geometry.
- Including spatial segmentation techniques.

Reflection

Overall, the project was a challenging but a rewarding experience. The project was successful in achieving many goals listed out in the requirement by implementing various features such as agents' steering behaviours, agent's states using simple rule-based decision making, collision detection and response, and physics-based simulations.

The agents were able to move in the terrain and reach the towers successfully. Also, the integration of physics-based simulations for the projectiles and collision detection added realism to the game.

One of the challenges faced during the project was implementing collision detection and response between different objects. It was a complex task that required a lot of experimentation and fine-tuning to get it working correctly. The collision detection between sphere-cylinder and sphere-cuboid was particularly challenging.

Another challenge faced was the integration of the different components of the game such as the agents, the projectiles, and the towers. The integration was done in a way that it could be updated at runtime using ImGui, this made the game more flexible and allowed for quick testing and debugging.

Overall, this project was a great opportunity to learn and apply various AI techniques and game development concepts. The project was successful in achieving not all but many goals and has provided a solid foundation for future improvements.

References

Real-time collision detection
Ericson, Christer

Game physics, 2nd ed
Eberly, David H

Physics for Game Developers
David M Bourg

Artificial intelligence for games, 2nd ed
Millington, Ian; Funge, John David