* Consider a hypothetical racing game where hundreds of cars race on a field. The updateRacers method shown below updates the cars and eliminates the ones that collide.

void UpdateRacers(float deltaTimeS, List<Racer> racers)

{

List<Racer> racersNeedingRemoved = new List<Racer>();

racersNeedingRemoved.Clear();

// Updates the racers that are alive

int racerIndex = 0;

for (racerIndex = 1; racerIndex <= 1000; racerIndex++)

{

if (racerIndex <= racers.Count)

{

if (racers[racerIndex - 1].IsAlive())

{

//Racer update takes milliseconds

racers[racerIndex - 1].Update(deltaTimeS \* 1000.0f);

}

}

}

// Collides

for (int racerIndex1 = 0; racerIndex1 < racers.Count; racerIndex1++)

{

for (int racerIndex2 = 0; racerIndex2 < racers.Count; racerIndex2++)

{

Racer racer1 = racers[racerIndex1];

Racer racer2 = racers[racerIndex2];

if (racerIndex1 != racerIndex2)

{

if (racer1.IsCollidable() && racer2.IsCollidable() && racer1.CollidesWith(racer2))

{

OnRacerExplodes(racer1);

racersNeedingRemoved.Add(racer1);

racersNeedingRemoved.Add(racer2);

}

}

}

}

// Gets the racers that are still alive

List<Racer> newRacerList = new List<Racer>();

for (racerIndex = 0; racerIndex != racers.Count; racerIndex++)

{

// check if this racer must be removed

if (racersNeedingRemoved.IndexOf(racers[racerIndex]) < 0)

{

newRacerList.Add(racers[racerIndex]);

}

}

// Get rid of all the exploded racers

for (racerIndex = 0; racerIndex != racersNeedingRemoved.Count; racerIndex++)

{

int foundRacerIndex = racers.IndexOf(racersNeedingRemoved[racerIndex]);

if (foundRacerIndex >= 0) // Check we've not removed this already!

{

racersNeedingRemoved[racerIndex].Destroy();

racers.Remove(racersNeedingRemoved[racerIndex]);

}

}

// Builds the list of remaining racers

racers.Clear();

for (racerIndex = 0; racerIndex < newRacerList.Count; racerIndex++)

{

racers.Add(newRacerList[racerIndex]);

}

for (racerIndex = 0; racerIndex < newRacerList.Count; racerIndex++)

{

newRacerList.RemoveAt(0);

}

}

Rewrite the method to improve its readability and performance without changing its behaviour.

Original update function checks collisions twice between the same pair of cars since the collider nested loop goes through the entire racers list. It also calls “onExplode” function several times for the same car, one per collision with any other car.

If we want to preserve the “onExplode” calls but reduce unnecessary collision checks we can change the index’s starting value of the nested loop.

The original update function also stores duplicated elements on the “racersNeedingRemoved” list. We will delete these duplicated elements before iterating through this list to improve performance.

void OptimizedUpdateRacers(float deltaTimeS, List<Racer> racers)

{

// Gets the racers that are still alive. Initialize by

List<Racer> racersNeedingRemoved = new List<Racer>();

// Updates the racers that are alive

for (int racerIndex = 0; racerIndex < racers.Count; racerIndex++)

{

if (racers[racerIndex].IsAlive())

{

//Racer update takes milliseconds

racers[racerIndex].UpdateRacer(deltaTimeS \* 1000.0f);

}

}

// Collides

//FIX: we only have to check the collision once between the same 2 racers

//the loops has less iterations

for (int racerIndex1 = 0; racerIndex1 < racers.Count; racerIndex1++)

{

Racer racer1 = racers[racerIndex1];

for (int racerIndex2 = racerIndex1 + 1; racerIndex2 < racers.Count; racerIndex2++)

{

Racer racer2 = racers[racerIndex2];

if (racer1.IsCollidable() && racer2.IsCollidable() && racer1.CollidesWith(racer2))

{

OnRacerExplodes(racer1);

//To preseerve the number of OnRacerExplodes calls we have to call it also with racer2 since we have reduced the nested loop iterations

OnRacerExplodes(racer2);

racersNeedingRemoved.Add(racer1);

racersNeedingRemoved.Add(racer2);

}

}

}

//Requiers "using System.Linq;". Removes duplicated elements,

racersNeedingRemoved = racersNeedingRemoved.Distinct().ToList();

// Get rid of all the exploded racers

racersNeedingRemoved.ForEach(i => i.Destroy());

//update racers List

racers.RemoveAll(i => racersNeedingRemoved.Contains(i));

}

* Describe further changes you would make if you could change its behaviour. Discuss your reasoning for making these changes.

**Following there’s a version of the update function changing its behavior to avoid using other racer list apart from the original racer one. In this version of the update function the destroy (or disable) call is executed inside the loop but doesn’t affect the overall order of execution.**

void NewBehaivourUpdateRacers(float deltaTimeS, List<Racer> racers)

{

// Updates the racers that are alive

for (int racerIndex = 0; racerIndex < racers.Count; racerIndex++)

{

if (racers[racerIndex].IsAlive())

{

//Racer update takes milliseconds

racers[racerIndex].UpdateRacer(deltaTimeS \* 1000.0f);

}

}

// Collides

//FIX: we only have to check the collision once between the same 2 racers

//the loops has less iterations

//Changing the loop to inverse order allow us to modify the iteration list while executing the loop

for (int racerIndex1 = racers.Count - 1; racerIndex1 >= 0; racerIndex1--)

{

Racer racer1 = racers[racerIndex1];

//indicates the need to modify the racer list

bool needsToRemove = false;

for (int racerIndex2 = racerIndex1 - 1; racerIndex2 >= 0; racerIndex2--)

{

Racer racer2 = racers[racerIndex2];

if (racer1.IsCollidable() && racer2.IsCollidable() && racer1.CollidesWith(racer2))

{

OnRacerExplodes(racer1);

OnRacerExplodes(racer2);

needsToRemove = true;

}

}

if (needsToRemove)

{

racers.Remove(racer1);

poolingSystem.ClearItems(racer1);

//The destroy is nor needed anymore if you use the pooling system but you can keep it if it implements additional logic

racer1.Destroy();

}

}

}

**Other changes that I’d suggest are to limit the scope of the update function to one single responsibility/purpose, as the SRP (single responsibility principle) indicates.**

**Currently the UpdateRacers function is in charge of updating the racers but also it checks the collisions and destroys the objects.**

**Since the racers are being destroyed and most likely instantiated again I have implemented a pooling system to save time enabling and disabling game objects and reusing instantiated racers. (This system is implemented in the unity project provided for the next part of the technical test)**

**If the amount of racers is high enough would be interesting to consider creating an entity component system for this matter (create a movement system which moves all the entities that have a transform and a velocity component, adding a collision system which checks collisions across all the elements that have a collision component and finally a destroy/pooling system to handle all the entities that can be recycled/destroyed)**

**If the entity component system is "over engineering" at least I would create different functions, one to check collisions, another one to move and a third one to destroy the racers (or disable if a pooling system is used).**

**I would extract explode, destroy logic outside the race. In the collision check loop I'd use racer1.Explode() (virtual function) instead of the “OnRacerExplode(racer1)”. This way we can extract the explosion logic to the racer class and make it easier to add different explosion types for each racer (i.e.).**

**Another important change to improve the performance could be to update only the racers that are rendered by the camera or the racers which affect the gameplay (a kind of interest management). This change would only be possible to implement depending on the nature of the game mechanics. In some racing games it is not possible to discriminate the racers according to this criteria.**

* Create a simple project in Unity 2021.3.10f1 using C# with the following requirements:  
    
  In the scene there are 100 objects that are spawned at random positions that can collide with a main game object.  
    
  The main game object moves at a constant speed, starting at (0, 0, 0), towards 3 points (P1, P2, P3) that can be changed by a designer.  
    
  At arrival at each point, the game object should begin movement toward the next point.  
    
  On arrival at the final point or if a collision is detected, the main game object should be removed from the scene, play a sound effect and show a particle effect.  
    
  Note that the game uses the physics system provided by Unity and the collisions are reported by Unity.