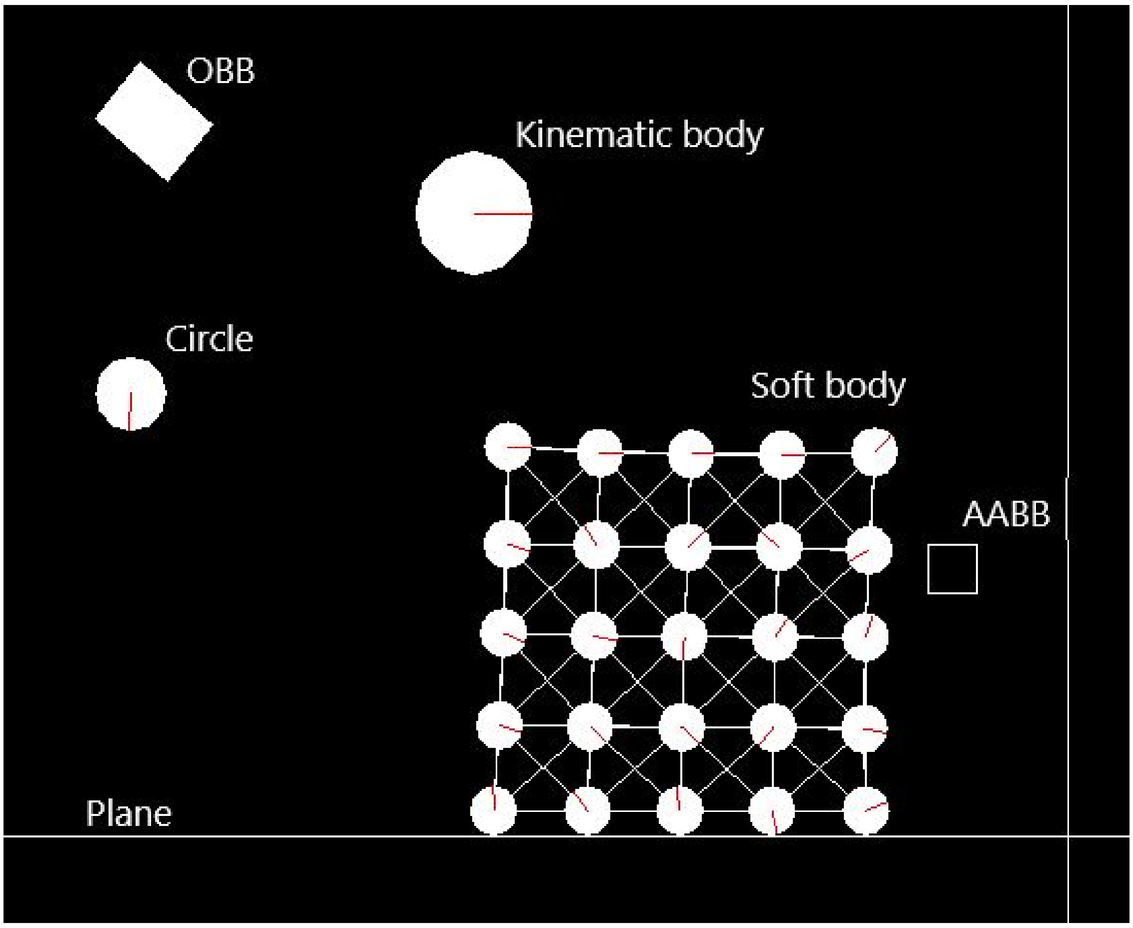
Custom Physics Project

Written by David F

Project description

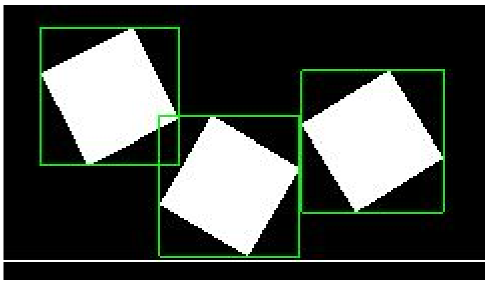
The application I have created is a working 2D C++ physics engine that can be implemented into a proper game engine. No third-party physics engine was used to aid the project.
It demonstrates a complex simulation containing multiple static and dynamic rigid bodies interacting together, as well as additional bodies such as joints, springs and soft bodies. All bodies are able to be added or removed from the simulation dynamically.
The engine includes four body shapes: circle, plane, axis-aligned bounding box (AABB), and oriented bounding box (OBB). Collision detection is performed using the spatial hashing method. The various body shapes are able to resolve collisions using linear forces and angular forces.



Improvements that could be made

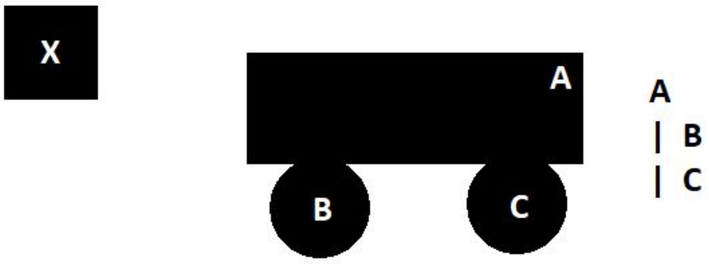
Optimise using a collision detection algorithm

In the physics engine presented through the tutorials, all actors in the scene check for collisions between all other actors in the scene. This is very unoptimised and can lead to reduced performance when there are many actors part of the simulation. I have addressed this in my code in two ways.

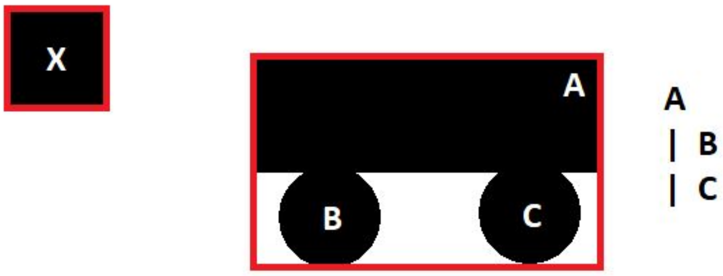


Firstly, by doing an AABB check between the actors before executing the more intensive collision detection algorithms. If the AABB check returns no overlap, then there's no reason to continue the check because there's no way the two actors could be overlapping.

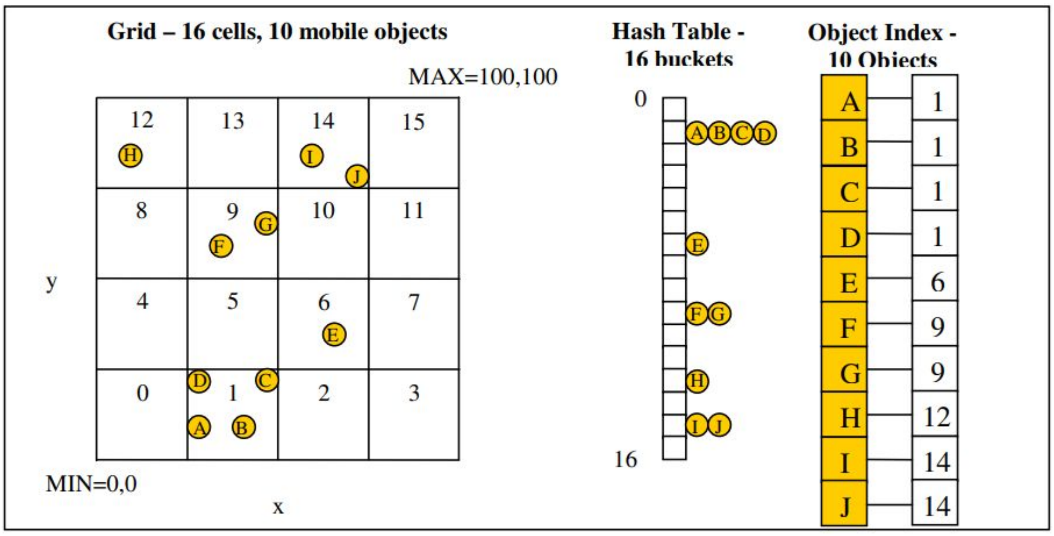
Furthermore, in a hierarchical game structure, with parents and children, the AABB check could be used to encompass each actor in the hierarchy, including their children. In the following example, object **A** is a parent to objects **B** and **C**. A collision check is being performed between **A, B** and **C** with object **X**.



To optimise collision detection, the engine can first create an AABB volume that surrounds **A** and its children. If this volume does not overlap the other object, **X**, then there is no way that objects **A, B** or **C** could be overlapping either - reducing three separate checks into one.



The second way I have improved collision performance is by implementing a collision detection algorithm, known as spatial hashing. This involves dividing the game scene up into a grid and allocating each actor to one or more cells in the grid based on their positions.



The above image[1] shows how the algorithm is implemented. The grid on the left contains ten objects (labelled A through J) alloted between 16 cells (buckets). A hash table is used to store which objects are part of which bucket. Each fixed update, the bucket is cleared and the objects are added back using their updated positions.

The advantage of this method is that objects need only check for collisions between other objects in its own cell. This is a significant improvement from checking against every other object in the scene. For example, object **H** in cell 12 will not need to do any collision detection on this frame. Object **D** will only need to check against objects **A, B** and **C**.

There are many improvements that can be made to this. One example includes only clearing dynamic bodies each frame, leaving the kinematic actors in the same cell as they can't change position. It would also help if bounding boxes were re-calculated only when the object has translated or rotated.

Overall, improving the physics engine with optimised collision detection methods makes it more suitable for use in a large-scale project.

Callback on collision detection

One disadvantage of the physics engine in its current state is that collisions cannot be resolved outside of the simulation. In other words, if it were to be used in a game project, there would be no way to determine when two actors have collided. The Unity3D engine uses C# reflection to callback on the MonoBehaviour instance when a collision has been detected, also passing through relevant information.
There are two main ways I can see of solving this in my engine: using function pointers to allow other sections of code to ‘hook’ on to a collision; or extending a rigidbody and using a virtual function.

Using the first approach would involve storing a vector of function pointers in the Rigidbody class. Code from outside of the class can then add their own function to the vector through a public method. Finally, when a collision is detected by the body, all of the function pointers in the vector are invoked. Below is code I have written to validate the approach.

```
protected:
      std::vector<std::function<void()>> m_callbacks;

public:
      void Rigidbody::HookOnCollision(std::function<void()> func)
      {
          m_callbacks.push_back(func);
      }

private:
      void Rigidbody::OnCollision()
      {
            // A collision is detected
            // Invoke function pointers in the callback vector
            for (std::function<void()> func : m_callbacks)
                  func();
      }
```

The code above only accepts methods with no arguments and no return type, however this could altered by replacing the function to something like:

```
std::function<void(Rigidbody*, const CollisionInfo&)>
```

I have tried implementing this in my project, however issues arise with attempting to pass in pointers to member functions. I’m sure there would be a way to solve it but I couldn’t find a solution.

My second idea is inspired by Unity3D's callback. The first step is to add a virtual function to the Rigidbody class that is invoked when a collision is detected. Secondly, a new class needs to be created that extends a Rigidbody class and implements the virtual function.

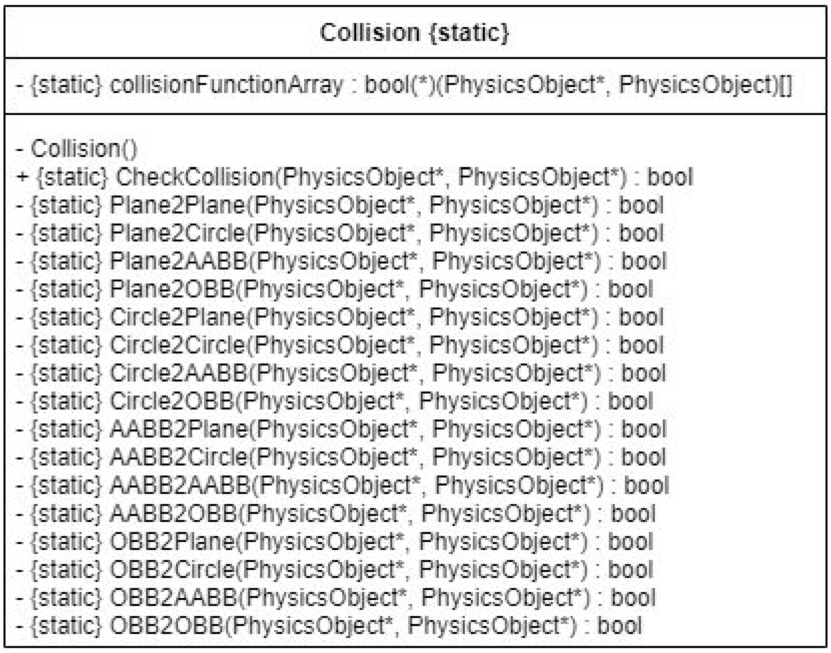
```
// In Rigidbody class
protected:
      virtual void OnCollision(Rigidbody*, const CollisionInfo&) {}

// New class that extends a Rigidbody class
class Ball : public Circle
{
public:
      Ball();
      ~Ball();

protected:
      virtual void OnCollision(Rigidbody*, const CollisionInfo&);
};
```

Collision detection functions moved off PhysicsScene

An improvement that could be made is moving the shape vs shape collision detection functions from the Physics Scene to their respective shape classes and/or a static Collision class. This way collisions can be checked at any time if needed without having to go through the Physics Scene class.



```
Collision {static}
- {static} collisionFunctionArray : bool(*)(PhysicsObject*, PhysicsObject)[]

- Collision()
+ {static} CheckCollision(PhysicsObject*, PhysicsObject*) : bool
- {static} Plane2Plane(PhysicsObject*, PhysicsObject*) : bool
- {static} Plane2Circle(PhysicsObject*, PhysicsObject*) : bool
- {static} Plane2AABB(PhysicsObject*, PhysicsObject*) : bool
- {static} Plane2OBB(PhysicsObject*, PhysicsObject*) : bool
- {static} Circle2Plane(PhysicsObject*, PhysicsObject*) : bool
- {static} Circle2Circle(PhysicsObject*, PhysicsObject*) : bool
- {static} Circle2AABB(PhysicsObject*, PhysicsObject*) : bool
- {static} Circle2OBB(PhysicsObject*, PhysicsObject*) : bool
- {static} AABB2Plane(PhysicsObject*, PhysicsObject*) : bool
- {static} AABB2Circle(PhysicsObject*, PhysicsObject*) : bool
- {static} AABB2AABB(PhysicsObject*, PhysicsObject*) : bool
- {static} AABB2OBB(PhysicsObject*, PhysicsObject*) : bool
- {static} OBB2Plane(PhysicsObject*, PhysicsObject*) : bool
- {static} OBB2Circle(PhysicsObject*, PhysicsObject*) : bool
- {static} OBB2AABB(PhysicsObject*, PhysicsObject*) : bool
- {static} OBB2OBB(PhysicsObject*, PhysicsObject*) : bool
```

The Collision class could then be improved upon further to contain other helpful methods such as raycasting or checking if a point lies within a shape.

Third-party libraries

aieBootstrap

https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap
aieBootstrap is a static library developed to aid in graphical real-time applications.

ImGui

https://github.com/ocornut/imgui
ImGui is an easy to use graphical user interface (GUI) library for C++. It has enabled me to quickly iterate and test new features through a click of a button. ImGui makes interacting with my program more intuitive, instead of trying to remember a collection of key shortcuts.

GLM

https://glm.g-truc.net/0.9.9/index.html
The OpenGL Maths (GLM) library is a header only C++ library for graphics software. It includes helpful classes such as vectors and matrices, as well as maths functions. It is used extensively throughout my project.

C++ Standard Library

https://en.cppreference.com/w/cpp/header

References

[1] Hastings, E., Mesit, J. and Guha, R. (2005). *Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing*. [online] Cs.ucf.edu. Available at: http://www.cs.ucf.edu/~jmesit/publications/scsc%202005.pdf [Accessed 24 Feb. 2020].

MacDonald, T. (2009). *Spatial Hashing*. [online] GameDev.net. Available at: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/spatial-hashing-r2697/ [Accessed 3 Mar. 2020].