2023

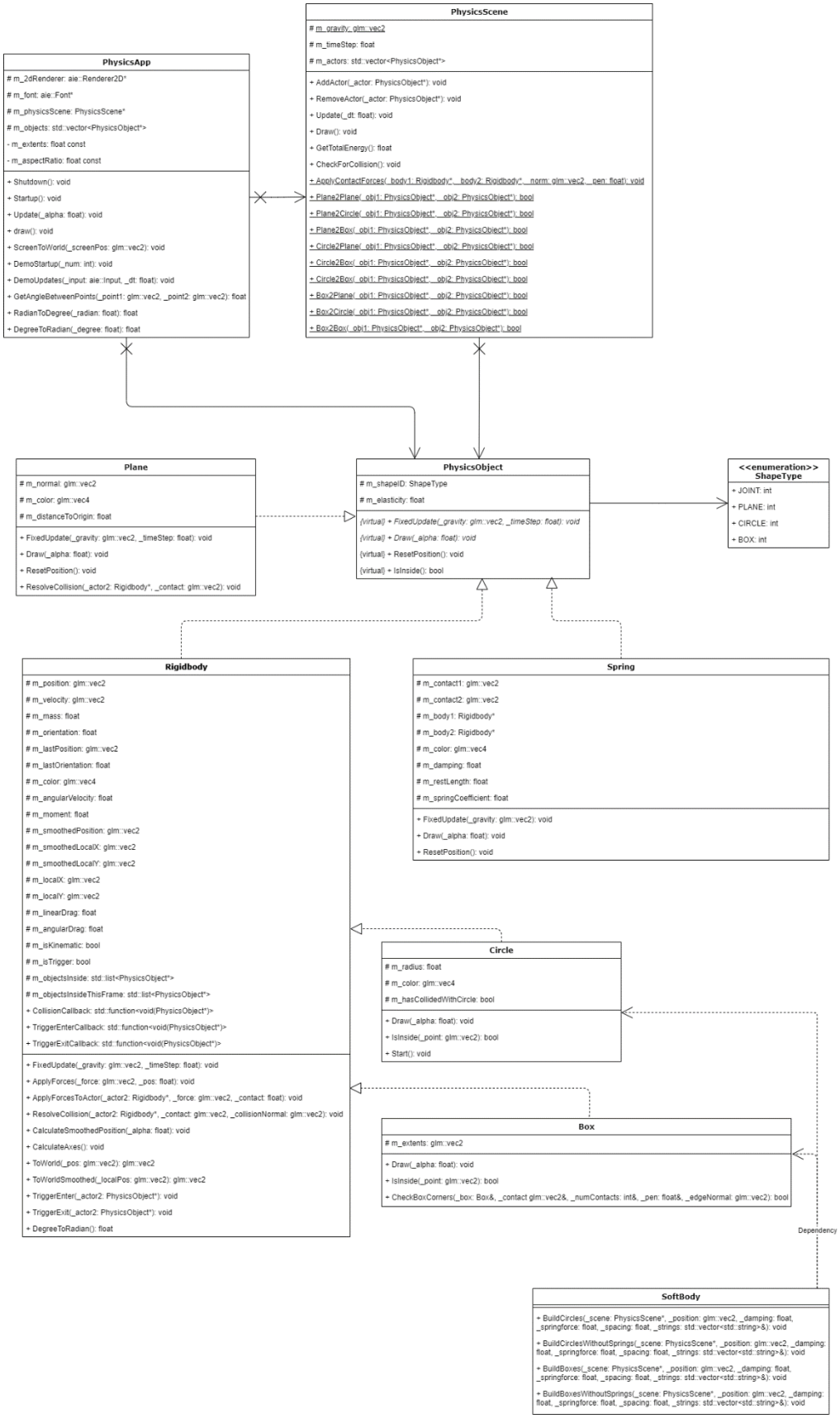
Custom Physics Documentation

EIGHT BALL MASTERS

MATTHEW CARVER

Contents

1.0 - Custom Physics Simulation Class Diagram



2.0 - Custom Physics Simulation Interactions

The Custom Physics Engine developed according to the tutorials is demonstrating a simplified physics simulation including force applications, simplified collisions, collision triggers, rotations and mass calculations.

The physical bodies can interact together as dynamic and static objects by virtue of the class structure of the bodies; all physical bodies are derived from the PhysicsObject class.

Static objects inherit directly from PhysicsObject and thus do not have the dynamic functionality for movement, mass, rotations etc. A Plane class is created for our static object.

Dynamic objects contain more functionality but are different in some fundamental ways which requires an intermediate class of inherit from. Rigidbody inherits from PhysicsObject and contains variables and functions that a common among all manner of primitive object interactions. A Circle and a Box class are the dynamic primitive classes created.

Dynamic and Static objects interactions are handled within the PhysicsScene class. The PhysicsScene checks for collisions amongst its actor list, identifies the PhysicsObject type, leverages a collision function array to assign a delegate for the appropriate collision function operation. The operation calculates and applies force to each actor (if they're dynamic) according to their properties.

Kinematic versions of the dynamic objects is possible by setting a variable that controls the mass of the objects and some functions, essentially making them "static" or kinematic.

3.0 - Custom Physics Simulation Potential Improvements

3.1 - Improvement #1 – Friction & Torque

One potential improvement envisioned for the custom physics library was friction with the purpose of implementing torque. Friction is the force resisting an objects' relative motion of solid surfaces, fluid layers and objects' sliding against each other. Friction acts as a force against an object in motion in the opposite direction to its relative motion. Torque is a measure of a force that can cause an object to rotate about an axis. Torque is what causes an object to acquire angular acceleration.

How it would have improved the project

These two features would have greatly improved the physics library in many ways. Using the visualisation as an example, the implementation of friction would have caused the balls on the table to slow and come to a stop in a more realistic and natural way that they do in its current state. It also would have represented a "wall bounce" more accurately as the friction force generated would alter the trajectory of the ball more realistically which adds depth to the interaction within the visualisation.

The implementation of torque would have produced more natural spin of the pool balls as they would be able to spin on any axis within their area. The torque coupled with friction would have greatly increased the realism of the physics library and visualisation.

These features were not implemented due to a combination of time constraints, complexity, and requirements. Upon revision and given ample time, friction and torque would be key features to include in the physics library which would allow for a more complex visualisation project in the future.

3.2 - Improvement #2 - Quadtrees

Another potential improvement to the custom physics library would revolve around efficiency by utilising quadtrees. A quadtree is a data structure that is used to divide a 2D region into more manageable parts. Like a binary tree with 2 nodes, a quadtree has four nodes that it manages. The 2D space starts as a single node and as objects are added into the 2D space, the parent node splits into 4 child nodes. This happens recursively as more objects are added.

A node with no children nodes in a quadtree is called a “leaf” node. Each leaf node handles the operations for the objects within its bounds which greatly increases efficiency of the simulation as a whole. It reduces the number of processes that occur in a physics library as any object within the area that have no physical forces are not processed. Any nodes with zero children are omitted from processing any physics functions, again, increasing the speed of the physics library.

A quadtree was not implemented into the data processing of the physics library due to the scope of the target visualisation and time constraint. The scope of the visualisation was low for Eight ball pool that the efficiency produced by a quadtree would have been negligible and invisible to the player. The time constraint of delivery played a factor into its omission into the library as development, testing and validation all would have extended beyond the delivery date for the visualisation.

4.0 - Visualised Game Using Your Custom Physics Simulation

The chosen visualisation utilising the custom physics simulation engine is Eight ball pool. Eight ball is a game with 16 balls on a table:

- 1 x White
- 1 x Black
- 7 x Solid Color
- 7 x Striped Color

The player must strike the white ball against the others on the table, abiding by the rules, sink their colored balls in the 6 pockets before sinking finally the black to win the game.

The project was created using the “Project2D” project as a template. The process of creation was as follows:

- Create a physics scene
- Creating a PoolBall class
- Loading balls into the game
- Creating wall “boxes”
- Creating pocket triggers
- Cue stick
- User Interface
- Implement rules

The game is set up by creating the boxes (walls), circles (pockets), and all the sixteen balls in play.

See an example of ball setup below:

```
m_whiteBall = new PoolBall(
    m_ballTex[15], // Texture
    WHITE, // Ball Type
    glm::vec2(_x: 91, _y: 0), // position
    glm::vec2(scalar: 0), // velocity
    m_poolBallMass, // mass
    m_poolBallRadius, // radius
    glm::vec4(_x: 1, _y: 1, _z: 1, _w: 1), // color
    _isKinematic: false, _isTrigger: false); // kinematic, trigger
m_physicsScene->AddActor(m_whiteBall);

// Place the balls into a triangle on the table
int ballsPlaced = 0;
float heightAdjuster = 0;
float poolDiameter = m_poolBallRadius * 2;
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 1 + i; j++)
    {
        int mid = (1 + i) / 2;
        if(mid == j)
        {
            heightAdjuster = 0;
        }
        else if(mid > j)
        {
            heightAdjuster = m_poolBallRadius + (mid - j) * m_poolBallRadius;
        }
        else
        {
            heightAdjuster = -m_poolBallRadius - (j - mid) * m_poolBallRadius;
        }
```

```
        if(ballsPlaced == 4)
        {
            // push back the 8 ball in next position
            m_blackBall = new PoolBall(
                m_ballTex[7], // Texture
                BLACK, // Ball Type
                    _position: glm::vec2( // position
                    _x: -60 - (i * poolDiameter),
                    _y: 0 + heightAdjuster),
                glm::vec2(scalar: 0), // velocity
                m_poolBallMass, // mass
                m_poolBallRadius, // radius
                glm::vec4(_x: 1, _y: 1, _z: 1, _w: 1), // color
                _isKinematic: false, _isTrigger: false); // kinematic, trigger
            m_physicsScene->AddActor(m_blackBall);
            ballsPlaced++;
            continue;
        }
        if(i % 2 != 0)
        {
            heightAdjuster -= m_poolBallRadius;
        }
        if (ballsPlaced % 2 != 1)
        {
            m_solidBalls.push_back(new PoolBall(
                m_ballTex[m_solidBalls.size()], // Texture
                SOLID, // Ball Type
                    _position: glm::vec2( // position
                    _x: -60 - (i * poolDiameter),
                    _y: 0 + heightAdjuster),
                glm::vec2(scalar: 0), // velocity
                m_poolBallMass, // mass
                m_poolBallRadius, // radius
                glm::vec4(_x: 1, _y: 1, _z: 1, _w: 1), // color
                _isKinematic: false, _isTrigger: false)); // kinematic, trigger
            m_physicsScene->AddActor(m_solidBalls[m_solidBalls.size()-1]);
        }
```

```
        else
        {
            m_stripedBalls.push_back(new PoolBall(
                m_ballTex[m_stripedBalls.size() + 8], // Texture
                STRIPED, // Ball Type
                    _position: glm::vec2( // position
                    _x: -60 - (i * poolDiameter),
                    _y: 0 + heightAdjuster),
                glm::vec2(scalar: 0), // velocity
                m_poolBallMass, // mass
                m_poolBallRadius, // radius
                glm::vec4(_x: 1, _y: 1, _z: 1, _w: 1), // color
                _isKinematic: false, _isTrigger: false)); // kinematic, trigger
            m_physicsScene->AddActor(m_stripedBalls[m_stripedBalls.size()-1]);
        }

        //m_physicsScene->AddActor(m_solidBalls[i]);
        ballsPlaced++;
    }
}

// Add all the balls to a single list
AddToBallList();
}
```

The game works by applying force to the white ball using the cue stick. The cue stick is measured against the white ball, the length is calculated, a force multiplier is applied and the direction vector is ascertained from the position of the mouse to the white ball. Using these variables, force is applied to the white ball.

The white ball, in turn, collides with the kinematic boxes that act as the walls of the table play area. Or a collision will occur with the other pool balls whereby the PhysicsScene object will check for and handle any collisions that occur between the actors in its PhysicsObject list. Finally, collisions also occur with the "pocket" circles that act as a trigger on collision with a PoolBall actor.

User interface elements are updated to both update the player and inform them on subsequent actions. Rules are checked when balls are sunk and/or balls are hit that do/do not comply with the rules. Pending the players actions, fouls are applied to the player and the other player gets an extra turn.

When all of the targeted balls that a player is attempting to sink are off the board, the players new target becomes the black ball. When a player targeting the black ball sinks it, they win the game assuming they do not sink the white ball in the same turn.

5.0 - Third Party Libraries

For this visualisation, no third-party libraries were used. The starter project provided, Bootstrap, contained enough functionality to develop the visualisation functionality, generate a renderer object, create a window, and display the sprite assets. If sound assets, visual effects, complex animation, or any other functionality was required to meet the expectations of the project, then a suitable library would have been researched and utilised.

6.0 - References

Szauer, G. (2017) *Game Physics Cookbook: Discover over 100 easy-to-follow recipes to help you implement efficient game physics and Collision Detection In Your Games*. Birmingham ; Mumbai: Packt.

Torque and angular momentum | physics library | science - khan academy (2016) *www.khanacademy.org*. Available at: https://www.khanacademy.org/science/physics/torque-angular-momentum (Accessed: February 20, 2023).

Friction (2023) *Wikipedia*. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Friction (Accessed: February 20, 2023).

Lambert, S. (2012) *Quick tip: Use quadtrees to detect likely collisions in 2d Space*, *Game Development Envato Tuts+*. Envato Tuts. Available at: https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374 (Accessed: February 20, 2023).