Created by Dragutin Sredojevic

Billiard Game Template

About

Billiard Game Template for Unity is an event-based billiard physics engine in which differential equations of motion are solved exactly, thus creating a simulation that conforms to reality. This method is much faster than conventional solvers used in physics engines as it does not have an explicit simulation step but rather a search is performed for the next event(state change) when the differential equation to be solved is updated. Differential equations are solved analytically so precision only depends on the model!

Support

Website: http://nitugard.pages.dev

Documentation: https://billiarddocs.pages.dev/

Forum: https://forum.unity.com/threads/billiard-template.1350635/

Discord: https://discord.gg/jEG3gCxhCK

Email: onedragutin@gmail.com

Getting Started

Check out demo scene located in **BilliardTemplate/Scenes/SampleScene.**

Instructions:

- Use the **mouse** to orient the camera.
- **Hold LMB** and move the mouse up and down to **zoom.**
- **Press S on the keyboard** to enter **aim mode**, once you position the cue stick **press S again** to enter **shot mode**. You can exit **shot mode** by pressing **RMB**.
- **Press V on the keyboard** to enter **view mode.**
- **Hold V on the keyboard** to move in **view mode** with the mouse.
- You can press **escape on the keyboard** at any time to toggle between **menu mode** and **view mode.**
- **Hold S on the keyboard** to show trajectory, press **Space on the keyboard** to strike the ball

Data

Simulation settings and other relevant data are stored inside **BilliardTemplate/Data/ using scriptable objects.**

Physics simulation parameters are stored in *PhysicsSolverConstants.asset* file that is located in **BilliardTemplate/Data/Solvers/** For accurate values check out: https://billiards.colostate.edu/faq/physics/physical-properties/

Input is processed using the new unity input system, and the input control asset is stored inside **BilliardTemplate/Data/Input** folder**.**

This template is using jobs and burst packages to greatly improve performance. Each time an event search is performed events are accumulated until the next event time surpasses **MinimumEventTime** or until the event array is **full.** Array capacity is determined by the **MaximumEventsPerStep** value. You can edit these values by editing the *BilliardPhysicsJobConstants.asset* file located in **BilliardTemplate/Data/Solvers/.**

Simulation

Events are processed inside the unity Update loop. During this step balls in the unity scene are transformed to match position and orientation towards the next event based on time elapsed.

Note: As it is not possible in general to integrate angular velocity over time, angular velocity is integrated during delta steps.

Billiard.cs simulation:

1. Start → Balls, Cushions and Holes inside the unity scene are collected and stored inside native memory via the following process:
    a. The unity component array is converted to the managed array
    b. The managed array is finally converted to a native array
2. Tick →
    a. If Stationary →
        i. Dispatches last event message
        ii. Runs the job that acquires the collection of events
        iii. Dispatches event messages except for the last event, which will be used for ``interpolation``
        iv. If a valid event exists dispatch the state change message and change the stationary value to false and reset the timer
        v. Dispatches state change message if the stationary value has changed during the previous frame
    b. If Not Stationary →

        i. Update timer by dt
        ii. Update managed and cached ball array to match the in-between state between the last state of the balls and the next event state of the balls based on the timer
        iii. Integrate rotation and cache the value
    c. Set the position and orientation of the unity ball array to match managed ball array
3. End → Native arrays are disposed

Input

Player input is processed based on the current mode(state) that the player is in. Possible modes include:

1. Aim mode – The mode in which the cue stick can be oriented
2. Shot mode – The player can precisely strike the ball in this mode
3. Spectator mode – Once the ball is struck player enters spectator mode, camera is now following cue ball
4. Menu mode - Prototype mode for the menu
5. View mode – Mode in which the user can freely orient and move the camera

User Input is gathered using the new unity input system while using simple abstraction. See **InputControl.cs**.

Each mode comes with a scriptable object file located in **BilliardTemplate/Data/Perspective/** folder.

Mode management is implemented using the finite-state machine and each registered mode is instantiated only once. The first mode in the array is the active one by default. Check out **ModeManager.cs** for more information.

Controllers

The two most important controllers are the Camera and Cue controllers. Both controllers have associated scriptable objects that store their data. Assets are located in **BilliardTemplate/Data/Controllers/** folder.

The camera controller provides methods to position and orient the camera while the cue controller is used to position and orient the cue and perform the **strike**!

User interface/Visual controllers:

- EightPoolInterfaceController
- OffsetController
- CrosshairController
- TrajectoryController

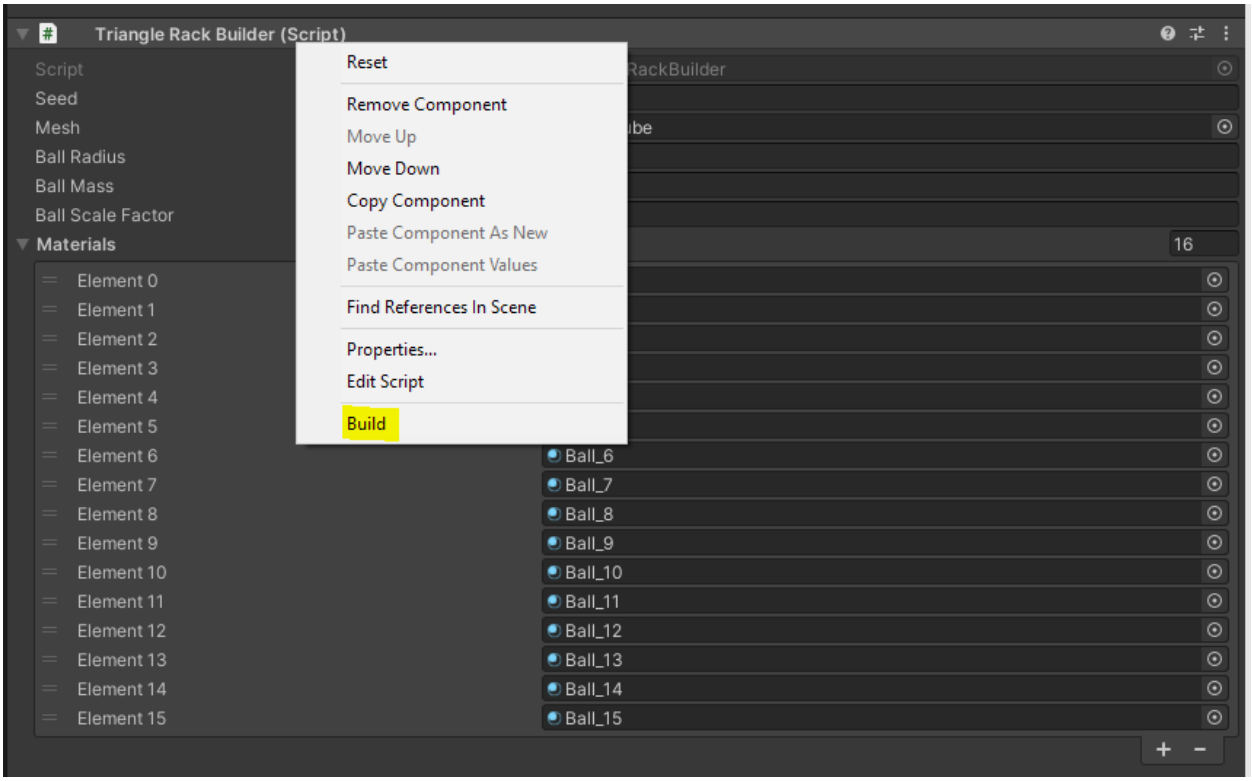
Builders

Building a billiard scene is not an easy task, that's where builders come in handy.

Eight pool builders:

- Table builder – When this builder is run It sets up a typical 8Pool billiard table collision plan in the unity scene
- Rack builder – When this builder is run it constructs a triangle of balls

To run a builder you must activate a specific build option from the context menu.



Console

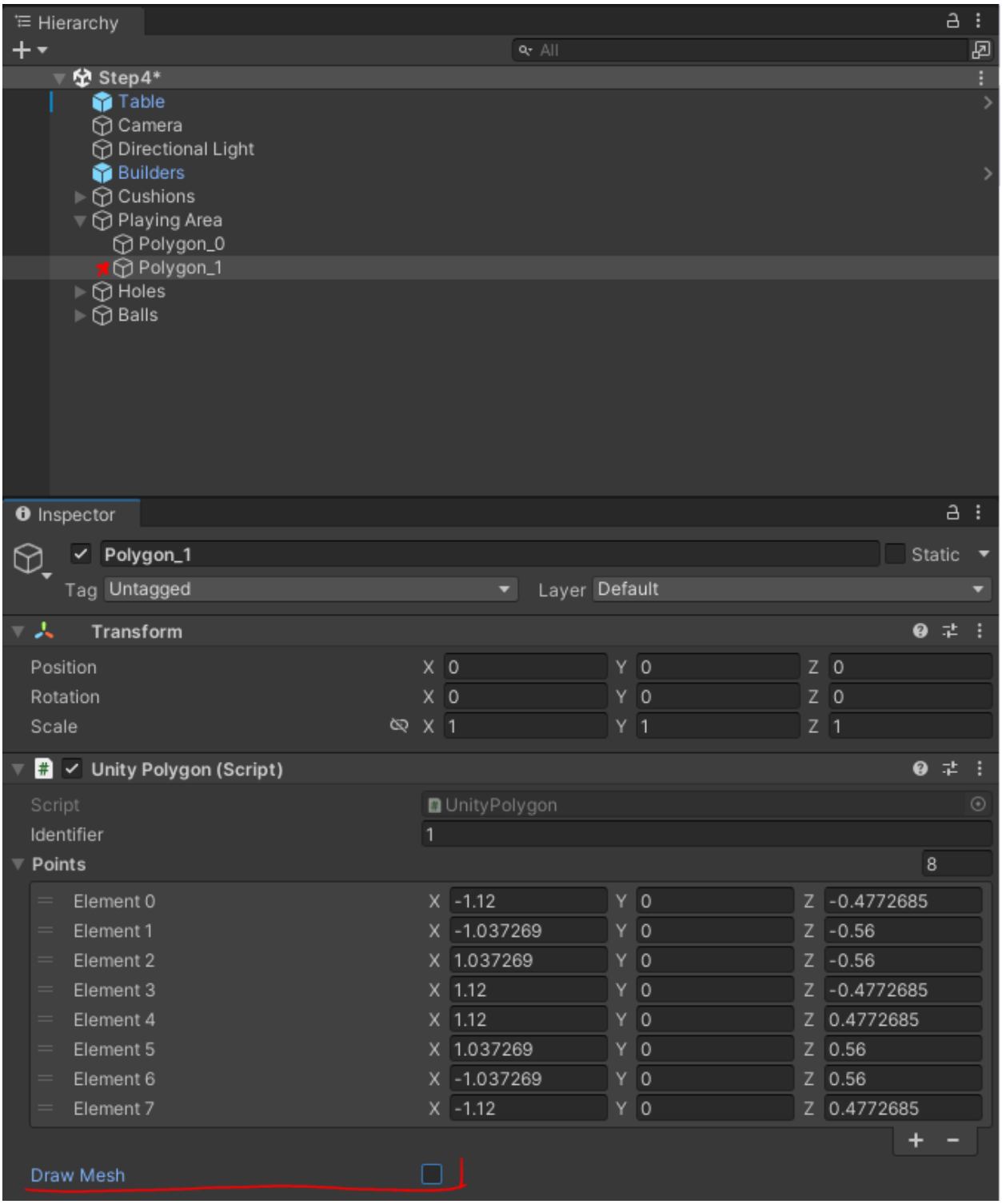
This project comes with a simple debug console. By default unity **Debug.Log** messages are shown. To remove the console simply remove IbcConsole from the scene.

From Scratch

To set up a new billiard scene from scratch follow the steps:

1. Create an empty scene, add a camera, lights, and the billiard table
2. Add Builder prefab in the scene and from the **EightPoolTableBuilder.cs** component context menu press build. This will generate collision shapes that define the table and the playing area. Notice the newly added game objects.

3. Change the parameters or manually reposition the shapes or the table to match the boundaries. Note that by default the values are more or less physically accurate and are measured in **meters**! Once you are finished you should get something like this:
4. Disable the playing surface visual.



5. Locate Builders prefab in the scene again and from the **TriangleRackBuilder.cs** component context menu press build. This will spawn balls inside a triangle. Reposition the balls. The final result looks:

6. Since the balls are small compared to standard unity measure units. Change the camera clipping distance to a minimum for both the in-game camera and the unity scene camera.
   Near: 0.01
   Far: 50
7. To get shadows working properly set the light bias close to zero, or zero.
8. Create a new game object, and rename it to “CueStick”. Add **UnityCue.cs** component to it and move the stick so it does not obstruct your view. You can also go ahead and replace the model.
9. Add a new game object, and rename it to “Billiard”. Add the following components:
   a. EightPoolBilliard.cs
   b. ModeManager.cs
   c. CameraController.cs
   d. CueController.cs
   e. EightPoolInterfaceController.cs
   f. PlayerInput.cs
10. Set up references in a straightforward manner, if you are unable to do so. Place the Billiard prefab instead and ignore the previous step.
11. [Optional]
    a. If you add menu mode, you need another camera with *MenuCamera* tag. Make sure it is active and enabled.
    b. Add trajectory manager to the scene and setup references similar to sample scene.
    c. Add offset indicator by placing the Offset prefab inside canvas.
12. **Play**.

Scene Objects

There are several objects relevant to billiard simulation, and each of them contains specific managed and unmanaged class representation. A managed class inherits from a unity mono behavior class while an unmanaged class is a simple struct data container.

All these types are collected inside **BilliardScene.cs**. Depending on their nature three cases are possible:

- BilliardUnityScene
- BilliardManagedScene
- BilliardNativeScene

Each object inside the class group is identified by its identifier, so make sure identifiers within a group are unique. For example, the white ball identifier created by default Rack Builder is 0, solids are 1-7, and stripe balls are 9-15 white the black ball uses identifier 8.

Cushion

A cushion object is a rigid body that has the shape of a polygon defined by the points and a height offset from the zero plane. Mass is assumed to be infinite.

The height offset indicates(in the case of 3d collision) where exactly is the top of the cushion.

Ball

A ball object is a uniform sphere, that has a mass of around *0.156kg* and a radius *0.028575m*, in the case of the 8Pool balls.

A model scale factor is used to scale the visual mesh representation(unity game object) to match the actual ball size.

Hole

A hole object is just a sphere that acts like a ghost during simulation. It can and will register events but it does not interact with balls.

On each hole object, there is a **UnityHoleDrop.cs** component attached, this script is used to fake a ball drop using interpolation between points based on the initial ball velocity.

Cue

The cue object holds relevant data such as mass, length, the radius of the tip, and impact parameters.

Polygon

A polygon is used to represent the playing surface and as such is useful to query which balls are inside the playing area.

This object has no role in the simulation otherwise.

Impact Laws

- Ball/Ball impact law uses a rigid body impact model with negligible friction assumption.
- Ball/Cushion impact law uses a rigid body impact model with a simple algebraic friction model
- Ball/CueTip impact law uses a rigid body impact model with negligible friction.

Friction

To a good approximation, the coefficient of friction between two surfaces is a constant value that is independent of the speeds of the two sliding bodies. If the ball is rolling dynamic friction is zero.
Static friction is usually larger than dynamic friction and occurs when two bodies have no relative velocity(are at rest), once the body starts to move dynamic friction takes over. The static friction force is accounted for during static analysis.

Typical Friction Range:

- Ball To Ball Friction: 0.03 – 0.08
- Ball To Slate Friction:
    - Rolling: 0.005 – 0.015
    - Sliding: 0.15 – 0.4
- CueTip To Ball: 0.6

Trajectory

Trajectory manager is super useful while debugging the game, it indicates where the balls will go if struck with specific velocity. It is precise and accurate but the performance is not that great and should be used with care.

Each time the cue stick transform is changed billiard state is integrated forward in time in specified time steps to obtain ball positions that lie in between events.

The trajectory manager is responsible for spawning trajectory controllers(ie. Trajectories) for each ball. It can also spawn ghost balls by providing a ghost ball prefab which would then spawn a ghost ball for each ball and for each time step increment provided that the ball moved sufficiently far enough.

The trajectory manager has its own billiard simulation that it manages so it does not interact in any way with the main simulation, this is done through BilliardState.cs.

Note: Since predicting the velocity of the player's strike is impossible, the demo is set up in such a way that the velocity used is based on the cue stick draw distance, and a space button is required to strike the ball with this velocity.

More

To get more information or if you require support, please do not hesitate to contact me.

You can also request GitHub source code access to get updates as soon as possible.

Thank you, Dragutin Sredojevic. Last update: November 5, 2022.