Rigid Body Dynamics Simulation Program

Natapat Nomnoei[1], Kaewmangkorn Choksathit[1], Tamonwan Nawacharoen[1*]

[1]*Princess Chulabhorn Science High School Chonburi, Nong Chak, Ban Bueng, Chon Buri, 20170 Thailand*

**E-mail: tamonwan@pccchon.ac.th*

Abstract

Nowadays, scientific progress is advancing rapidly, resulting in the emergence of countless educational media. While the educational media can greatly assist learners, they also come with limitations. This project aims to develop a single computer program to simulate the motion of 3-dimensional rigid body objects in classical mechanics. During testing, the team simulated 3 scenarios using the program to determine the precision and accuracy of the computed values. The tested parameters include the following: distance travelled in the Y-axis in free fall for 1 second, final velocity along the Y-axis during free fall after 1 second, and the mechanical energy with the reference level plane Y = 0.5 during seconds 0-1, with the downward acceleration due to gravity being at a constant 9.8100 meters per second squared without linear drag. The results reveal that the program-computed values correlate, but do not perfectly align with, the formula-computed ones. The distance travelled computed with the program depicts a constant 0.584 percent error, while the final velocity shows one of 0.01 percent. Furthermore, the mechanical energy diverges away from the formula-computed ones in the negative direction over time. This inconsistency with the incorrectly computed parameter may be caused by the limitations of using the Rigidbody Component in Unity to calculate. On the other hand, testing of the performance impact, which is carried out by evaluating the effects of the number of objects within a scene at a given time on the FPS value, shows a great capability to handle large amounts of game objects. Nevertheless, this project faces various obstacles during development, for instance: client-side device compatibility, limited user-editable variables, and the inability to save and load created scenes. As an improvement, more variables should be introduced, along with the function to save and load user-generated scenes

Keywords : Rigid Body, Component, Program-computed Value, Formula-computed Value

Introduction

Nowadays, scientific progress is advancing rapidly, resulting in the emergence of countless educational media. Part of these educational media includes various programs, simulating linear motion, projectile motion, circular motion, rotational motion, momentum and collisions, etc. While the growing pool of educational media can greatly assist learners in visualizing the material, these media, on their own, also have their limitations. For instance, the variety and freedom in simulating given scenarios. This leaves the user

needing to switch between programs after each simulation. As a result, the work needed to learn increased.

To address this challenge, the project team decided to develop a computer program to simulate the motion of rigid body objects using Unity for students and the interested learners. The learners can study the behavior of rigid body objects in linear motion, projectile motion, circular motion, rotational motion, momentum and collisions, and forces through the program's ability to draft a free body diagram and show each parameter of the object, for example, velocity, angle, and forces at a given moment, to ease the difficulties in studying Mechanics.

Methodology

The project team for the Rigid Body Dynamics Simulation Program, during development, followed the following methodology:

1) Obtaining Hardware and Software

1. A computer, along with an adequate internet connection.
2. Unity (Version 2023.2.20f1)

2) Programming

1. Install and open Unity 2023.2.20f1 and create a new project
2. Study the following:
    2.1. Basic Mechanics
    2.2. Linear Motion
    2.3. Projectile Motion
    2.4. Circular Motion
    2.5. Rotational Motion
    2.6. Momentum and Collisions
3. Design the framework and interface
    3.1. Framework

The program is divided into 4 main parts, controlled by the following objects:

3.1.1. Game Controller:

Controls camera movement, cursor position, and Inspector Tab states.

3.1.2. Input Controller:

Controls user input and the exchange of information between the Inspector Tab and Objects.

3.1.3. Object Controller:

Controls the selection of Objects.

3.1.4. Inspector:

Lives in the user UI, contains input fields to receive user input and exchange the received values with the Input Controller.

3.2. Main User Interface

3.2.1. Panel:

Contains the UI elements, including the Inspector Tab, timer, etc.

3.2.2. Inspector Tab:

1) Rigidbody Toggle: Controls the on or off rigid body state of the selected object.

2) Mass Parameter: Displays and receives mass input.

3) Position Parameter: Displays and receives 3-dimensional position input.

4) Rotation Parameter: Displays and receives 3-dimensional Euler rotation input.

5) Velocity Parameter: Displays and receives 3-dimensional velocity input.

6) Angular Velocity Parameter: Displays and receives 3-dimensional angular velocity input.

7) Energy Parameter: Displays Kinetic, Potential, and Mechanical energy.

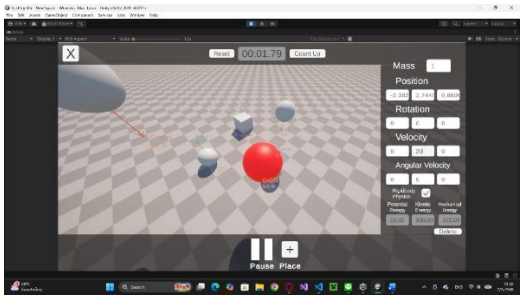

Fig. 1 Shows the Game Panel and Inspector Tab on the left-hand side of the screen

4. Develop the Program Using C# Programming Language

The following are the .css C# scripts that were used in the program:

4.1. Mouselock controller

A component of the Game Controller, which controls the visibility and mouse-locking. The cursor is locked to a fixed position and its visibility is set to invisible by default, and by holding the Alt key, the cursor becomes briefly visible and unconfined.

```
// Update is called once per frame
void Update()
{
    HandleMouseCursor();
}
public void HandleMouseCursor()
{
    // Handing Cursor Visibility
    // Inspector On
    if (InspectorTab.GetComponent<InspectorTab>().active)
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
    // Alt held
    else if (Input.GetKey(KeyCode.LeftAlt))
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
    // Alt not held
    else
    {
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;
    }
    if (ObjectController.GetComponent<InspectObject>().SelectedObjects.Count > 0)
    {
        Cursor.visible = true;
    }
}
```

Fig. 2 Shows the code used to control the mouse locking behavior.

4.2. Camera Movement

The Camera Movement module is another component of the Program Controller. It enables the camera to continuously follow the user by calculating the relative position between the player and the camera using a positional

offset vector. This approach establishes smooth and responsive follow behavior while preserving a consistent viewing perspective.

```
public class FollowPlayer : MonoBehaviour
{
    public Transform Leader;
    public UnityEngine.Vector3 offset;
    public UnityEngine.Vector3 angleOffset;

    // Start is called before the first frame update
    void Start()
    {
        offset = Leader.transform.position - transform.position;
        angleOffset = Leader.transform.eulerAngles - transform.eulerAngles;
    }

    // Update is called once per frame
    void Update()
    {
        transform.position = Leader.transform.position + offset;
        transform.eulerAngles = Leader.transform.eulerAngles + 0 * angleOffset;
    }
}
```

Fig. 3 Shows the code for the camera movement

4.3. User Interface Controller

The User Interface Controller, integrated within the Program Controller, manages the visibility and interaction of the graphical user interface, including the activation and deactivation of Edit Mode. The interface is organized into three operational levels:

Level 0: The user interface remains hidden. This is the default state when the program is launched. Transitioning to another level is only possible after an object has been selected.

Level 1: Displays the user interface containing the physical properties and parameters of the selected object.

Level 2: Expands the Inspector Panel when the user scrolls upward twice consecutively, providing access to additional configuration options.

```
public void HandleScreenPosition()
{
    // Is inspecting an object
    if (ObjectController.GetComponent<InspectObject>().SelectedObjects.Count > 0 &&
    ObjectController.GetComponent<InspectObject>().SelectedObjects[0] != null)
    {
        active = true;
    }
    else
    {
        level = 0;
        active = false;
    }


    uiElementToMove.anchoredPosition = Vector2.MoveTowards
        (uiElementToMove.anchoredPosition,
        new Vector2(offScreenPos.x - (level * pixelPerLevel), uiElementToMove.anchoredPosition.y),
        UIMoveSpeed * Time.deltaTime);
}

public void OnSelectObject()
{
    level = 1;
}
```

Fig. 4 Shows the code for controlling the Inspector Tab

```
public void HandleInputFields()
{
    while (ActiveInputFields.Count < AllInputFields.Count)
    {
        ActiveInputFields.Add(null);
    }

    if (level == 2)
    {
        for (int i = 0; i < AllInputFields.Count; i++)
        {
            if (ActiveInputFields[i] == null)
            {
                GameObject inputGroup = Instantiate(AllInputFields[i]);

                ActiveInputFields[i] = inputGroup;

                inputGroup.transform.SetParent(inspectorTransform, false);

                RectTransform rt = inputGroup.GetComponent<RectTransform>();
                rt.anchoredPosition = new Vector2(2.5f + (165 * i), 150 - (55 * i));
            }
        }
    }
    else
    {
        for (int i = 0; i < ActiveInputFields.Count; i++)
        {
            if (ActiveInputFields[i] != null)
            {
                Destroy(ActiveInputFields[i]);
                ActiveInputFields[i] = null;
                ActiveInputFields.RemoveAt(i);
            }
        }
    }
}
```

Fig. 5 Shows the code for syncing Inspector values toggling Edit Mode

4.4. Variable Input

The Variable Input module is part of the Input Controller and is responsible for synchronizing data between the selected object and the user interface. Physical parameters are retrieved from the object and displayed within the interface, while user modifications are immediately applied back to the object. Each editable parameter is assigned to a dedicated case statement—for example, Case 1 corresponds to mass, whereas Case 2 represents the object's X-coordinate position. In addition, the module performs real-time calculations of the object's mechanical energy.

```
public void SyncInspector()
{
    GameObject obj = ObjectController.GetComponent<InspectObject>().SelectedObjects[0];

    Rigidbody rb = SelectedObject.GetComponent<Rigidbody>();
    float gravity = Physics.gravity.y;
    float VelMagnitude = SelectedObject.GetComponent<PlacedObjects>().linearVelocity.magnitude;

    Collider col = SelectedObject.GetComponent<Collider>();
    float YCenterOffset = 0;

    if (col is SphereCollider sphere)
    {
        YCenterOffset = sphere.radius;
    }
    else if (col is BoxCollider box)
    {
        YCenterOffset = box.size.y / 2;
    }
    else if (col is CapsuleCollider capsule)
    {
        YCenterOffset = capsule.height / 2;
    }
    else
    {
        YCenterOffset = 0;
    }

    float PotentialEnergy = rb.mass * -gravity * (SelectedObject.transform.position.y - YCenterOffset);
    float KineticEnergy = 0.5f * rb.mass * Mathf.Pow(VelMagnitude, 2);
    float MechanicalEnergy = PotentialEnergy + KineticEnergy;
```

Fig. 6 Shows the code for computing parameters

```
    for (int i = 0; i < InspectorInputFields.Count; i++)
    {
        switch (i)
        {
            case 0:
                Parameters[0] = SelectedObject.GetComponent<Rigidbody>().mass;
                InspectorPreviewTMP[0].text = SelectedObject.
                GetComponent<Rigidbody>().mass.ToString();
                break;

            case 1:
                Parameters[1] = SelectedObject.transform.position.x;
                InspectorPreviewTMP[1].text = SelectedObject.transform.position.x.ToString();
                break;

            case 2:
                Parameters[2] = SelectedObject.transform.position.y;
                InspectorPreviewTMP[2].text = SelectedObject.transform.position.y.ToString();
                break;

            case 3:
                Parameters[3] = SelectedObject.transform.position.z;
                InspectorPreviewTMP[3].text = SelectedObject.transform.position.z.ToString();
                break;

            case 4:
                Parameters[4] = SelectedObject.transform.eulerAngles.x;
                InspectorPreviewTMP[4].text = SelectedObject.transform.eulerAngles.x.ToString();
                break;
            case 5:
                Parameters[5] = SelectedObject.transform.eulerAngles.y;
                InspectorPreviewTMP[5].text = SelectedObject.transform.eulerAngles.y.ToString();
                break;
            case 6:
                Parameters[6] = SelectedObject.transform.eulerAngles.z;
                InspectorPreviewTMP[6].text = SelectedObject.transform.eulerAngles.z.ToString();
                break;
```

Fig. 7 Shows the code for displaying Inspector values

```
public void ParameterChange(string fieldName)
{
    List<GameObject> objs = ObjectController.GetComponent<InspectObject>().SelectedObjects;

    switch (fieldName)
    {
        case "mass":
            UpdateValue(fieldName, InspectorInputFields[0].text);
            break;

        case "x":
            UpdateValue(fieldName, InspectorInputFields[1].text);
            break;

        case "y":
            UpdateValue(fieldName, InspectorInputFields[2].text);
            break;

        case "z":
            UpdateValue(fieldName, InspectorInputFields[3].text);
            break;

        case "rotX":
            UpdateValue(fieldName, InspectorInputFields[4].text);
            break;

        case "rotY":
            UpdateValue(fieldName, InspectorInputFields[5].text);
            break;

        case "rotZ":
            UpdateValue(fieldName, InspectorInputFields[6].text);
            break;

        case "velX":
            UpdateValue(fieldName, InspectorInputFields[7].text);
            break;

        case "velY":
            UpdateValue(fieldName, InspectorInputFields[8].text);
            break;
```

Fig. 8 Shows the code for syncing Inspector values

4.5. Object Selection

The Object Selection module, which belongs to the Input Controller, enables users to select or deselect objects through ray casting. When the user clicks within the simulation environment, a ray is projected from the camera toward the cursor position. If the ray intersects a rigid body object, the corresponding selection state is updated accordingly.

```
    }
        }
            }

            {
            \\ Move it to the object it point to
        {
        \\ Object found
    {
}
```

Fig. 9 Shows the code for Ray-casting

```
public void SelectObject(GameObject objectToSelect, bool deselect)
{
    if (objectToSelect != null)
    {
        if (!deselect)
        {
            // Set the Closest Object as the Selected Item
            SelectedObjects.Add(objectToSelect);

            // Flag this Object as ALREADY SELECTED in its script
            objectToSelect.GetComponent<PlacedObjects>().Selected = true;
        }
        else
        {
            // Set the Closest Object as the Selected Item
            SelectedObjects.Remove(objectToSelect);

            // Flag this Object as ALREADY SELECTED in its script
            objectToSelect.GetComponent<PlacedObjects>().Selected = false;
        }
    }
}
```

Fig. 10 Shows the code for controlling object selection

4.6. User Movement

The User Movement component, incorporated into the Player Movement system, enables free navigation throughout the simulation environment. User input is continuously processed to control the movement of the player object, allowing intuitive exploration of the virtual workspace.

```
void HandleMovement()
{
    float speed = MoveSpeed;

    // Sprinting
    if (Input.GetKey(KeyCode.LeftShift))
        speed *= SprintMultiplier;

    // Get WASD input
    float x = Input.GetAxis("Horizontal"); // A/D
    float z = Input.GetAxis("Vertical");   // W/S

    // Direction relative to camera
    UnityEngine.Vector3 moveDir = (transform.forward * z + transform.right * x).normalized;

    // Apply movement
    transform.position += moveDir * speed * Time.deltaTime;

    // Ascend (Space) / Descend (Left Ctrl)
    if (Input.GetKey(KeyCode.Space))
        transform.position += UnityEngine.Vector3.up * AscendDescendSpeed * Time.deltaTime;

    if (Input.GetKey(KeyCode.LeftControl))
        transform.position += UnityEngine.Vector3.down * AscendDescendSpeed * Time.deltaTime;
}
```

Fig. 11 Shows the code for the user's movement

```
void HandleMouseLook()
{
    // Get mouse delta
    float mouseX = 0;
    float mouseY = 0;

    if (Cursor.lockState == CursorLockMode.Locked)
    {
        mouseX = Input.GetAxis("Mouse X") * LookSensitivity;
        mouseY = Input.GetAxis("Mouse Y") * LookSensitivity;
    }

    if (false && InspectorTab.GetComponent<InspectorTab>().active && !EventSystem.current.IsPointerOverGameObject())
    {
        if (Input.mousePosition.x > (Screen.width - 160) * 3 / 4)
        {
            mouseX = 10 * LookSensitivity * Time.deltaTime;
        }

        if (Input.mousePosition.x < (Screen.width - 160) / 4)
        {
            mouseX = -10 * LookSensitivity * Time.deltaTime;
        }
        mouseY = Input.GetAxis("Mouse Y") * LookSensitivity;
    }

    rotY += mouseX;
    rotX -= mouseY;
    rotX = Mathf.Clamp(rotX, MinLookX, MaxLookX);

    // Apply rotation

    transform.rotation = UnityEngine.Quaternion.Euler(rotX, rotY, 0f);
}
}
```

Fig. 12 Shows the code for the user turning

4.7. Draw Arrows

The Draw Arrows module is attached to each simulated object and is responsible for performing relevant physical calculations while visualizing important vector quantities. Using Unity's Line Renderer, the module renders vectors representing linear velocity, angular velocity, applied force, and applied torque, thereby providing users with a clear graphical interpretation of the object's physical state.

```
6 references
public void Draw(Vector3 origin, Vector3 force, Color color,
                 float arrowScale, float headSize,
                 string labelName, bool showLabel, bool showMag,
                 bool isAngular = false)
{
    if (force.magnitude < 0.001f) { Root.SetActive(false); return; }
    Root.SetActive(true);

    // ผลคูณ log เพื่อไม่ให้ลูกศรยาวเกินไปเมื่อแรงมาก
    Vector3 scaled = force.normalized * (Mathf.Log(force.magnitude + 1f) * arrowScale);
    Vector3 end = origin + scaled;
    Vector3 dir = scaled.normalized;

    // ก้าน
    SetLR(shaft, origin, end, color);

    // คำนวณ right vector สำหรับหัวลูกศร
    Vector3 right;
    if (isAngular)
    {
        // หันหมุนตามเวลาเพื่อแสดง rotation direction
        float angle = force.magnitude * Mathf.Rad2Deg * Time.time;
        Quaternion spin = Quaternion.AngleAxis(angle, dir);
        right = Vector3.Cross(dir, spin * Vector3.up).normalized;
    }
    else
    {
        right = Vector3.Cross(dir, Vector3.up).normalized;
    }
    if (right.sqrMagnitude < 0.0001f)
        right = Vector3.Cross(dir, Vector3.forward).normalized;
```

Fig. 13 Shows the code for drawing the forces, velocity, and torque vectors

4.8. Object Replication

The Object Replication module forms part of the Object Controller and allows users to generate additional rigid body objects dynamically. New objects are created by instantiating predefined object prefabs stored within the project's Assets directory.

```
public void ChangeObjectBy(int indexChange)
{
    int index = (PlaceableObjects.IndexOf(ObjectToPlace) + indexChange) % PlaceableObjects.Count;
    if (index < 0)
    {
        index = PlaceableObjects.Count + index;
    }

    ObjectToPlace = PlaceableObjects[index];
    PreviewEulerY = 0;
}
public void PlaceObject(GameObject objectToPlace, GameObject placer, float placeDistance = 2, float eulerRotationY = 0)
{
    UnityEngine.Vector3 spawnPos = placer.transform.position + (placer.transform.forward * placeDistance);
    Instantiate(objectToPlace, spawnPos, Quaternion.Euler(0, eulerRotationY, 0));
}
```

Fig. 14 Shows the code for placing a Game Object

4.9. Object Behavior

The Object Behavior module is attached directly to each simulated object and governs its physical state and dynamic behavior. It manages the object's status information while interacting with Unity's Rigidbody component, which defines essential physical properties and enables realistic motion. Through this module, objects respond appropriately to physical quantities such as velocity, acceleration, force, and torque, thereby ensuring behavior consistent with the laws of classical mechanics

```
public void HandleSelected()
{
    // Changing Colors of Object
    if (Inspecting)
    {
        GetComponent<Renderer>().material = InspectingMaterial;
    }
    else if (Selected)
    {
        GetComponent<Renderer>().material = SelectedMaterial;
    }
    else if (MouseOn)
    {
        GetComponent<Renderer>().material = MouseOnMaterial;
    }
    else
    {
        GetComponent<Renderer>().material = material;
    }

    if (editingValues)
    {
        GetComponent<Renderer>().material = Red;
    }
}
```

Fig. 15 Shows the code for setting the materials for the Game Object

```
public void HandleRigidbodyLock()
{
    if (
        !rigidbodyActive ||
        ObjectController.GetComponent<InspectObject>().RigidbodyPause ||
        editingValues
    )
    {
        if (GetComponent<Rigidbody>().constraints == RigidbodyConstraints.None)
        {
            GetComponent<Rigidbody>().velocity = linearVelocity;
            GetComponent<Rigidbody>().angularVelocity = angularVelocity;
        }

        GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll;
    }
    else
    {
        if (GetComponent<Rigidbody>().constraints == RigidbodyConstraints.FreezeAll)
        {
            GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None;

            GetComponent<Rigidbody>().velocity = linearVelocity;
            GetComponent<Rigidbody>().angularVelocity = angularVelocity;
        }
        else
        {
            GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None;
        }

        linearVelocity = GetComponent<Rigidbody>().velocity;
        angularVelocity = GetComponent<Rigidbody>().angularVelocity;
    }
}
```

Fig. 16 Shows the code for setting the values for the Rigidbody Component

5. Simulation Validation

The completed simulation program was evaluated by conducting a series of representative physical scenarios designed to assess its numerical accuracy, functional reliability, and computational performance.

6. Program Refinement

Following the validation process, the identified limitations and implementation issues were systematically addressed. The program was subsequently refined through code optimization, algorithmic improvements, and interface enhancements to improve its accuracy, stability, computational efficiency, and overall user experience.

1.3.

5. Flow Chart

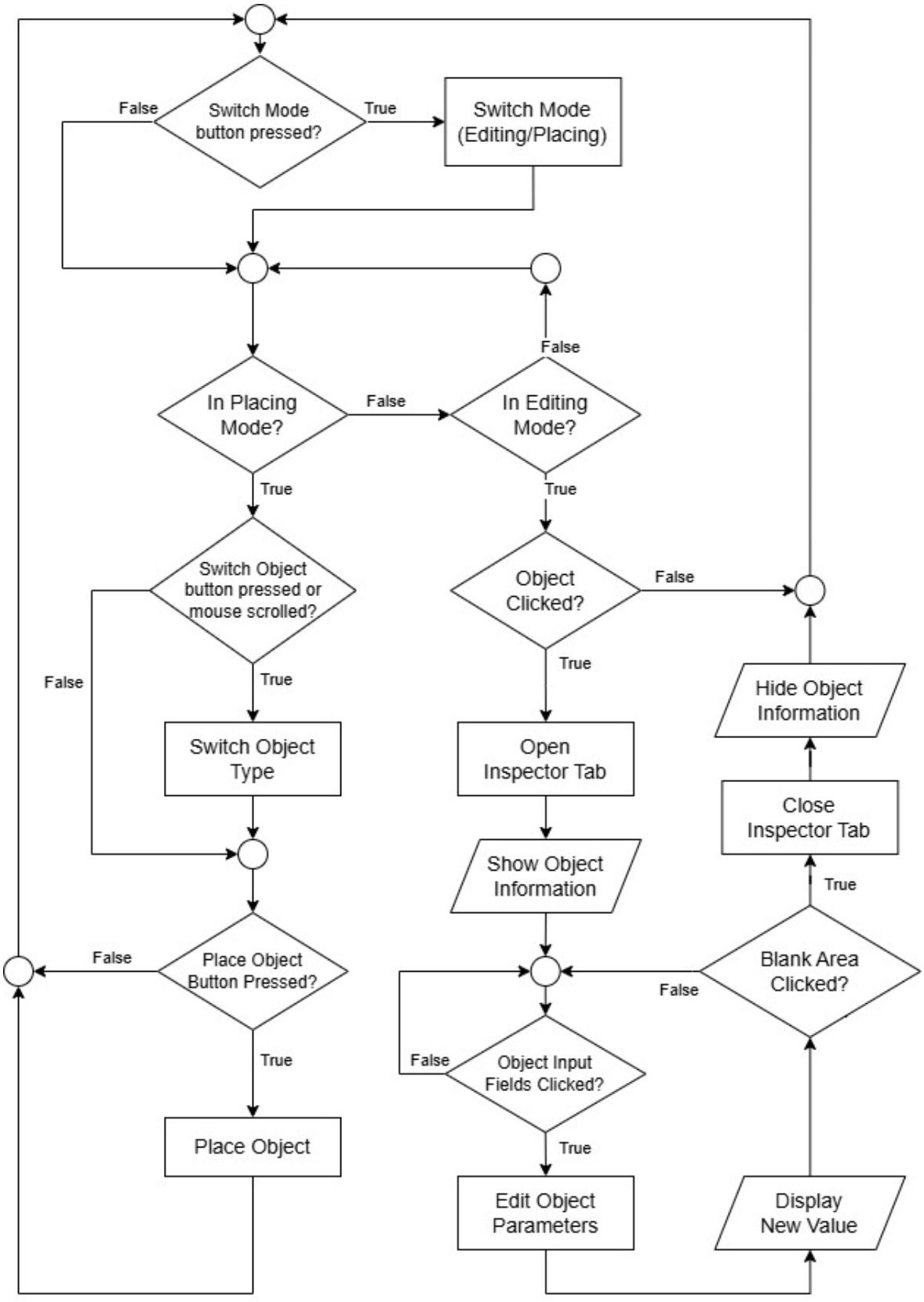

Fig. 17 Shows a flow chart representing the thought process of the program

Results and Discussions

The Rigid Body Dynamics Simulation Program was developed to accurately simulate both translational and rotational motion based on the fundamental principles of classical mechanics. To evaluate the program's reliability and performance, a series of validation experiments was conducted, with the assessment divided into three categories.

1) Evaluating the Accuracy of the Program-Computed Values

The accuracy of the simulation was evaluated by comparing the results generated by the program with theoretical values obtained through analytical calculations. Three independent experimental scenarios were designed and tested, with each scenario repeated three times, resulting in a total of nine trials. The objective was to assess the consistency and numerical accuracy of the simulation under different physical conditions.

1.1 Scenario 1

The first validation scenario simulated the free fall of a rigid body released from a height of 10 meters above the reference plane (Y = 0) with zero initial velocity. The displacement of the object after 1 second was recorded by the simulation and subsequently compared with the theoretical value calculated using the kinematic equations of motion under constant gravitational acceleration. This comparison was performed to evaluate the accuracy of the program's implementation of translational dynamics.

Using physical formulae, the computed value is expressed by the following:

$$s = 4.9050\ m$$

Where s represents the distance covered along the Y-axis

Table 1 Shows the final Y position, distance travelled in the Y-axis, and error of the program-computed values with three repetitions.

Values	Program-computed y_f (m)	s (m) = 10 - y_f	Flat Error	s Error Percentage
1	5.0211	4.9789	0.0739	1.51
2	5.0211	4.9789	0.0739	1.51
3	5.0211	4.9789	0.0739	1.51
4	5.0211	4.9789	0.0739	1.51
mean	5.0211	4.9789	0.0739	1.51

1.2 Scenario 2

The second validation scenario simulated the free fall of a rigid body released from a height of 10 meters above the reference plane (Y = 0) with zero initial velocity. The objective was to determine the object's velocity along the Y-axis after 1 second of motion and compare the simulated results with the corresponding theoretical values derived from the equations of kinematics.

The formula-computed value is expressed by the following:

$$v_f = -9.810\ m/s$$

Where v_f represents the final velocity along the Y-axis

Table 2 Shows the final velocity along the Y-axis, and the error of the program-computed values with three repetitions.

Values	Program-computed v (m/s)	Flat Error	v Error Percentage
1	-9.809	0.001	0.01
2	-9.809	0.001	0.01
3	-9.809	0.001	0.01
4	-9.809	0.001	0.01
mean	-9.809	0.001	0.01

1.3 Scenario 3

The third validation scenario simulated the free fall of a 1 kg rigid body released from a height of 19.5 meters above the reference plane (Y = 0.5, corresponding to an initial Y-coordinate of 20), with zero initial velocity. Throughout the simulation, the relationship between mechanical energy and time was recorded and represented graphically to evaluate the program's ability to preserve total mechanical energy during free-fall motion.

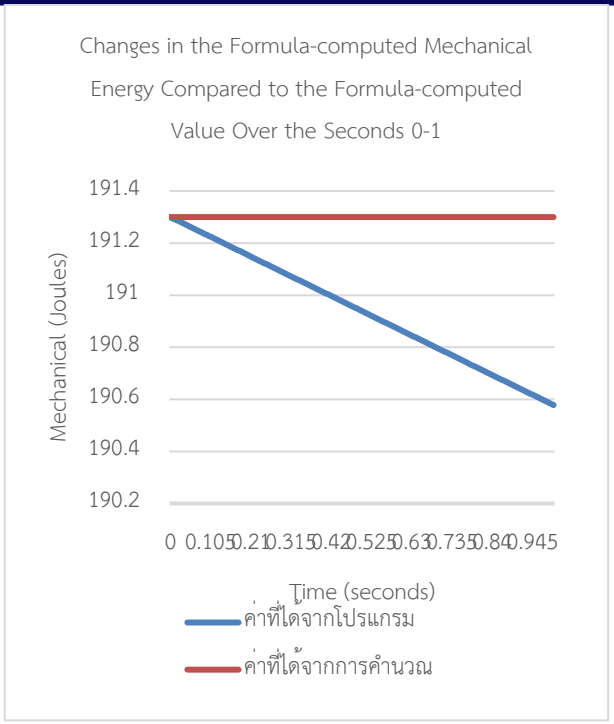

Fig. 18 A line plot showing the changes in the formula-computed mechanical energy compared to the formula-computed value over the seconds 0-1.

As illustrated in the figure above, the simulated mechanical energy agrees closely with the theoretical value at the initial instant, prior to the commencement of the measurement period. As the simulation progresses, however, the total mechanical energy exhibits a gradual decline and increasingly deviates from the theoretical prediction. This behavior can be attributed to the accumulation of numerical errors introduced in the displacement and velocity calculations observed in Scenarios 1 and 2. Such discrepancies are inherent to the real-time numerical integration performed by Unity's Rigidbody component and therefore represent an unavoidable limitation of the simulation framework rather than an error in the underlying physical model.

2) Evaluating the Impact of Object Count on the Performance

The computational performance of the simulation was evaluated by examining the effect of increasing the number of simulated objects on the program's rendering performance. The test was conducted by launching the simulation and incrementally adding rigid body objects, one at a time. The program's frames per second (FPS) were recorded after every additional 25 objects, covering a range from 0 to 125 objects. The experiment was repeated three times to ensure the reliability and reproducibility of the results.

Table 3 Shows the FPS value at different amounts of objects with three repetitions

Objects	0	25	50	75	100	125
1	60	60	60	57	54	49
2	60	60	58	57	52	50
3	60	60	59	56	53	48
Mean	60	60	59	56.67	53	49

The performance evaluation revealed a gradual decline in FPS as the number of simulated objects increased. When the scene contained 0–25 objects, the simulation consistently maintained an average frame rate of 60 FPS, representing the maximum performance achieved during the experiment. This indicates that the program is capable of handling this workload efficiently while providing smooth and responsive real-time visualization.

As the number of objects increased to 50, a noticeable reduction in frame rate was observed, indicating a greater computational burden on the simulation engine. At the maximum test condition of 125 objects, the average frame rate decreased to 49 FPS. Although the program remained fully functional under this workload, users began to experience perceptible rendering latency and reduced visual smoothness. Nevertheless, the observed performance demonstrates that the simulation maintains an acceptable level of responsiveness even under relatively demanding computational conditions.

Conclusions

This project successfully developed a Rigid Body Dynamics Simulation Program, a physics-based simulation application designed to facilitate the study and visualization of rigid body dynamics for students, educators, and individuals interested in physics. The simulation was developed using the Unity game engine, enabling an interactive environment in which users can observe and analyze physical phenomena in real time.

Performance evaluation demonstrated that the simulation achieved a high level of accuracy. The program was validated through three experimental scenarios, including a free-fall simulation in which an object was released from a height of 10 meters above the reference plane (Y = 0) with an initial velocity of zero. The object's velocity along the Y-axis after one second was measured and compared over four repeated trials. The results indicated a high degree of consistency and reliability, confirming the accuracy of the simulation.

In terms of computational performance, the program exhibited excellent efficiency. Benchmark testing showed that the simulation maintained stable and responsive performance

while handling between 0 and 50 rigid body objects simultaneously within a single scene.

Acknowledgements

The seminar organizing team would like to express its deepest gratitude to the Faculty of Science, Chulalongkorn University, for granting access to the thesis that served as the foundation of this seminar. The invaluable academic resources and insights provided by the Faculty have played a crucial role in the successful completion of this scientific seminar.

The team also wishes to extend its sincere appreciation to the teachers of the Science and Technology Learning Area at Princess Chulabhorn Science High School Chonburi, for their unwavering guidance, constructive advice, and continuous support throughout the preparation of this seminar. Their expertise and encouragement have been instrumental in enriching our understanding and ensuring the successful accomplishment of this work.

References

Dare, W. A., & Harold, S. S. (1983). Physics for Engineering and Science. Schaum's Outline Series in Engineering. New York, NY: McGraw-Hill.

Laohudomphan, C. (2017). Dessert Physics, Volume 1 (7th ed.). Bangkok, Thailand: Chulalongkorn University Press.

Somphong, J. (2005). University Physics 1 (6th ed.). Bangkok, Thailand: Chulalongkorn University Press.

Unity Technologies. (2023). Unity User Manual: Scripting in Unity for Experienced Programmers. Retrieved August 2, 2025, from https://unity.com/how-to/programming-unity

W3Schools. (n.d.). C# Get Started. Retrieved September 8, 2025, from https://www.w3schools.com/cs/cs_getstarted.php