RV Educational Institutions ®

RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

TITLE: ThreadRush: Virtual City Thread Joyride

Submitted by

Name	USN
SAMARTH G	1RV22CS174
RISHAB R	1RV22CS415
SACHIN ANNIGERI	1RV22CS174
SANTHRUPTH H R	1RV22CS416

Computer Science and Engineering 2023-2024

INDEX:

Problem Statement:

Students studying computer science or related fields face difficulty in understanding thread management concepts in operating systems. This occurs due to the abstract nature of the concepts and the lack of practical demonstrations, leading to challenges in grasping these concepts during the early stages of learning. The issue is commonly observed in educational environments where thread management is taught, affecting students who rely on traditional teaching methods that offer limited visual representations. To address this, there is a need for innovative teaching approaches and interactive learning experiences that provide students with opportunities to explore, experiment, and apply thread management concepts in a more engaging and practical manner.

Introduction to Problem:

Learning thread management in operating systems can be a bumpy road for students, especially in the beginning. Here's a breakdown of the common challenges:

Abstraction Overload: Thread management deals with concepts that are inherently abstract. Threads are lightweight processes that run concurrently, sharing resources within a program. This can be difficult to visualize without practical demonstrations.

Limited Practical Application: Traditional teaching methods often rely on theoretical explanations and code examples. While valuable, these methods lack the opportunity for students to "get their hands dirty" and experiment with threads themselves. This can lead to difficulty grasping the practical implications of the concepts.

Visual Learning Gap: Textbooks and lectures often lack engaging visual representations of thread behavior. Students who learn best visually might struggle to connect abstract concepts to real-world scenarios.

Impact on Students: These challenges can lead to several issues for students:

Difficulty Grasping Core Concepts: Students might struggle to understand how threads work, how they interact, and how to manage them effectively.

Limited Problem-Solving Skills: Without practical experience, students might find it difficult to apply their knowledge to solve real-world thread management problems.

Reduced Motivation: Traditional methods can become tedious, leading to decreased student engagement and motivation to learn.
The Need for a New Approach:

Traditional teaching methods can leave students feeling lost in the maze of thread management. To address these challenges, there's a need for innovative approaches that offer:
Interactive Learning: Experiences that allow students to explore and experiment with thread behavior, solidifying their understanding.
Practical Application: Opportunities to apply thread management concepts to simulated or real-world scenarios.
Engaging Visualizations: Visual representations that bring abstract concepts to life, making them more relatable for students who learn best visually.

VR Thread City: A Potential Solution:

VR Thread City concept has the potential to address these challenges by providing an interactive and visually engaging learning experience. By placing students in a virtual environment where threads are represented as cars navigating a city, you can offer them the opportunity to:

Visualize Thread Behavior: The city metaphor can represent critical concepts like synchronization (traffic lights) and resource sharing (shared roads).
Experiment with Scenarios: Students can create and manipulate threads (cars), observing how they interact and impact the virtual environment.

Apply Thread Management Principles: They can practice managing threads to achieve specific goals within the city, promoting problem-solving skills.
This VR approach has the potential to overcome the limitations of traditional methods and make learning thread management a more engaging and effective experience for students.

Relevant Operating System Concept and APIs used:

Operating System Concepts:

- **Threads:** Students will be able to visualize threads as cars navigating the virtual city. Each car represents a lightweight process within the program, executing concurrently with other threads.
- **Processes:** The VR environment itself can be considered a single process, spawning multiple threads (cars) to manage different aspects of the city (e.g., traffic control, building construction).

- **Synchronization:** Traffic lights within the city can represent synchronization mechanisms (e.g., mutexes, semaphores) that ensure proper thread coordination. Cars approaching an intersection (critical section) would need to acquire a "green light" (lock) to proceed safely, preventing collisions (race conditions).

- **Resource Sharing:** Roads and shared spaces within the city represent resources that threads (cars) must access concurrently. Students can observe potential issues like deadlocks if multiple threads (cars) try to occupy the same road segment (resource) at the same time.

Potential APIs(Exact APIs syntax as in OS is not used but the program written for the API behaves almost in the same way:

- **Thread Creation and Management:** APIs for creating new threads (spawning cars), terminating threads (removing cars), and manipulating thread priorities.

- **Synchronization Primitives**: APIs for implementing synchronization mechanisms like mutexes, semaphores, or condition variables (represented by traffic lights, yield signs, etc.)

- **Shared Memory Access:** APIs for accessing and modifying shared memory regions (e.g., simulating data exchange between threads at specific locations within the city).

Design of solution to problem:

Concept:

City Background:

This is a great visual metaphor for representing a computer system. Various dropdowns on road can represent resources and next location to move, and roads can represent common resources shared.

- **Cars as Threads:** Each car signifies a thread, with its destination representing the thread's function. Movement speed can represent the thread's priority.
- **Traffic Rules:** These represent various OS concepts like synchronization

(e.g., traffic lights) and resource management (e.g., limited parking spaces).

- **User Input:** Users can spawn new cars (threads), change their priorities, and introduce obstacles (e.g., road closures) to simulate different scenarios.

Implementation:

Unity and C#: This is a great choice for VR development. Utilize Unity's physics engine to simulate car movement and handle collisions. C# scripting will control car behavior and implement thread concepts.

Car Movement:

- Defined different car prefabs based on priority levels (e.g., color, size).
- Implemented pathfinding algorithms to guide cars towards their destinations.
- Used Unity's physics engine for realistic movement and collision detection.

- **Traffic Lights:** Implementing traffic lights at intersections to represent synchronization primitives like mutexes or semaphores. Cars need to acquire the "green light" (resource) to proceed.

- **Yield Signs:** Used yield signs at specific locations to represent situations where threads voluntarily relinquish control. Cars slow down and allow others to pass based on programmed logic.

User Interaction:

- Provide a VR controller or hand tracking to allow users to:
- Spawn new cars (threads) with different priorities.
- Change a car's destination dynamically (thread context switching).
- Introduce obstacles (e.g., road closures) to simulate system events like I/O operations.

Output Display:

- Dashboard: Create a virtual dashboard within the VR environment displaying:
- Information about each car (thread ID, priority, status).
- Resource utilization (e.g., parking space occupancy).

- Performance metrics (e.g., average wait time at traffic lights).
- Text Pop-ups: Display contextual messages above cars to highlight specific OS concepts being demonstrated (e.g., "Acquiring resource lock", "Context switch").

Source code:

Traffic light function:

```
void OnTriggerEnter(Collider other)
   {
      if (other.CompareTag("TrafficLight"))
      {
         TrafficLight light = other.GetComponent<TrafficLight>();  // Assuming a TrafficLight script exists
         if (light.isRed)
         {
            GetComponent<Rigidbody>().velocity = Vector3.zero;
         }
      }
      else if (other.CompareTag("YieldSign"))
      {

   }
}
```

Thread or Car multiplying function:

```
public void MultiplyCar()
   {

      GameObject newCar = Instantiate(this.gameObject, transform.position + Vector3.right * 2.0f, Quaternion.identity);



      newCar.GetComponent<CarController>().destination = someDestination;  // Assuming this is set elsewhere
      newCar.GetComponent<CarController>().speed = Random.Range(3.0f, 6.0f); // Randomize speed for variation
```

```
    }
}
```

Move to particular destination:

```
if (destination != null)
        {
            // Calculate the direction vector towards the destination
            Vector3 direction = destination.position - transform.position;

            // Use Vector3.MoveTowards with clamping for smoother movement
            transform.position = Vector3.MoveTowards(transform.position,
destination.position, speed * Time.deltaTime);

            // Optional: Rotate the car to face the direction of movement
            transform.rotation = Quaternion.LookRotation(direction);
        }
    }
```

Process synchronization(attached to traffic lights):

```
        public class TrafficLight : MonoBehaviour
        {
            public enum LightState { Red, Yellow, Green };  // Enum for traffic light
        states
            public LightState currentState;              // Current state of the traffic
        light
            public float redLightDuration = 5.0f;         // Duration of red light
            public float yellowLightDuration = 1.0f;      // Duration of yellow light
        (optional)

            private float timer = 0.0f;                // Timer for state duration

            void Start()
            {
                currentState = LightState.Red;           // Start with red light
                ChangeLightColor();                    // Set initial light color
            }
```

```
    void Update()
    {
        timer += Time.deltaTime;                // Update timer

        if (currentState == LightState.Red && timer >= redLightDuration)
        {
            ChangeState(LightState.Yellow);       // Transition to yellow
(optional)
        }
        else if ((currentState == LightState.Yellow && timer >=
redLightDuration + yellowLightDuration) ||
                 currentState == LightState.Green && timer >=
greenLightDuration)
        {
            ChangeState(LightState.Red);          // Transition back to red
        }
    }

    void ChangeState(LightState newState)
    {
        currentState = newState;
        timer = 0.0f;                         // Reset timer
        ChangeLightColor();                   // Update light color based on state
    }

    void ChangeLightColor()
    {
        // Access the Light component attached to this GameObject and set its
color based on currentState
        Light lightComponent = GetComponent<Light>();
        switch (currentState)
        {
            case LightState.Red:
                lightComponent.color = Color.red;
                break;
            case LightState.Yellow:
                lightComponent.color = Color.yellow;
                break;
            case LightState.Green:
                lightComponent.color = Color.green;
```

```
            break;
        }
    }
}
```

Car movement by user:

```
public class CarController : MonoBehaviour
{
    private float horizontalInput, verticalInput;
    private float currentSteerAngle, currentbreakForce;
    private bool isBreaking;

    // Settings
    [SerializeField] private float motorForce, breakForce, maxSteerAngle;

    // Wheel Colliders
    [SerializeField] private WheelCollider frontLeftWheelCollider,
frontRightWheelCollider;
    [SerializeField] private WheelCollider rearLeftWheelCollider,
rearRightWheelCollider;

    // Wheels
    [SerializeField] private Transform frontLeftWheelTransform,
frontRightWheelTransform;
    [SerializeField] private Transform rearLeftWheelTransform,
rearRightWheelTransform;

    private void FixedUpdate() {
        GetInput();
        HandleMotor();
        HandleSteering();
        UpdateWheels();
    }

    private void GetInput() {
        // Steering Input
        horizontalInput = Input.GetAxis("Horizontal");
```

```
        // Acceleration Input
        verticalInput = Input.GetAxis("Vertical");

        // Breaking Input
        isBreaking = Input.GetKey(KeyCode.Space);
    }

    private void HandleMotor() {
        frontLeftWheelCollider.motorTorque = verticalInput * motorForce;
        frontRightWheelCollider.motorTorque = verticalInput * motorForce;
        currentbreakForce = isBreaking ? breakForce : 0f;
        ApplyBreaking();
    }

    private void ApplyBreaking() {
        frontRightWheelCollider.brakeTorque = currentbreakForce;
        frontLeftWheelCollider.brakeTorque = currentbreakForce;
        rearLeftWheelCollider.brakeTorque = currentbreakForce;
        rearRightWheelCollider.brakeTorque = currentbreakForce;
    }

    private void HandleSteering() {
        currentSteerAngle = maxSteerAngle * horizontalInput;
        frontLeftWheelCollider.steerAngle = currentSteerAngle;
        frontRightWheelCollider.steerAngle = currentSteerAngle;
    }

    private void UpdateWheels() {
        UpdateSingleWheel(frontLeftWheelCollider,
frontLeftWheelTransform);
        UpdateSingleWheel(frontRightWheelCollider,
frontRightWheelTransform);
        UpdateSingleWheel(rearRightWheelCollider,
rearRightWheelTransform);
        UpdateSingleWheel(rearLeftWheelCollider,
rearLeftWheelTransform);
    }

    private void UpdateSingleWheel(WheelCollider wheelCollider,
Transform wheelTransform) {
        Vector3 pos;
```

```
        Quaternion rot;
        wheelCollider.GetWorldPose(out pos, out rot);
        wheelTransform.rotation = rot;
        wheelTransform.position = pos;
    }
}
```

New movements:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CarController : MonoBehaviour
{
    private float horizontalInput, verticalInput;
    private float currentSteerAngle, currentbreakForce;
    private bool isBreaking;

    // Settings
    [SerializeField] private float motorForce, breakForce, maxSteerAngle;

    // Wheel Colliders
    [SerializeField] private WheelCollider frontLeftWheelCollider,
frontRightWheelCollider;
    [SerializeField] private WheelCollider rearLeftWheelCollider,
rearRightWheelCollider;

    // Wheels
    [SerializeField] private Transform frontLeftWheelTransform,
frontRightWheelTransform;
    [SerializeField] private Transform rearLeftWheelTransform,
rearRightWheelTransform;

    private void FixedUpdate() {
        GetInput();
        HandleMotor();
        HandleSteering();
        UpdateWheels();
    }
```

```
    private void GetInput() {
        // Steering Input
        horizontalInput = Input.GetAxis("Horizontal");

        // Acceleration Input
        verticalInput = Input.GetAxis("Vertical");

        // Breaking Input
        isBreaking = Input.GetKey(KeyCode.Space);
    }

    private void HandleMotor() {
        frontLeftWheelCollider.motorTorque = verticalInput * motorForce;
        frontRightWheelCollider.motorTorque = verticalInput * motorForce;
        currentbreakForce = isBreaking ? breakForce : 0f;
        ApplyBreaking();
    }

    private void ApplyBreaking() {
        frontRightWheelCollider.brakeTorque = currentbreakForce;
        frontLeftWheelCollider.brakeTorque = currentbreakForce;
        rearLeftWheelCollider.brakeTorque = currentbreakForce;
        rearRightWheelCollider.brakeTorque = currentbreakForce;
    }

    private void HandleSteering() {
        currentSteerAngle = maxSteerAngle * horizontalInput;
        frontLeftWheelCollider.steerAngle = currentSteerAngle;
        frontRightWheelCollider.steerAngle = currentSteerAngle;
    }

    private void UpdateWheels() {
        UpdateSingleWheel(frontLeftWheelCollider,
frontLeftWheelTransform);
        UpdateSingleWheel(frontRightWheelCollider,
frontRightWheelTransform);
        UpdateSingleWheel(rearRightWheelCollider,
rearRightWheelTransform);
        UpdateSingleWheel(rearLeftWheelCollider,
rearLeftWheelTransform);
    }
```

```
        private void UpdateSingleWheel(WheelCollider wheelCollider,
    Transform wheelTransform) {
            Vector3 pos;
            Quaternion rot;
            wheelCollider.GetWorldPose(out pos, out rot);
            wheelTransform.rotation = rot;
            wheelTransform.position = pos;
        }
    }
```

Collectable Components:

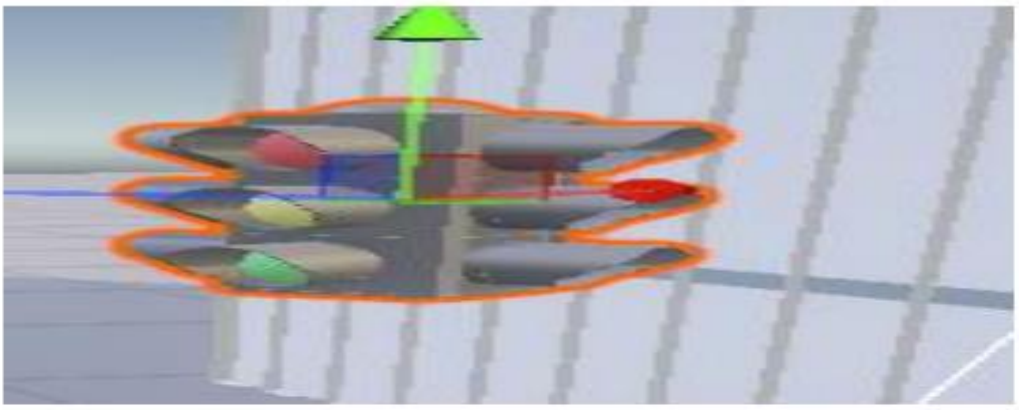
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CollectableComponent : MonoBehaviour
{
    public string itemType;
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

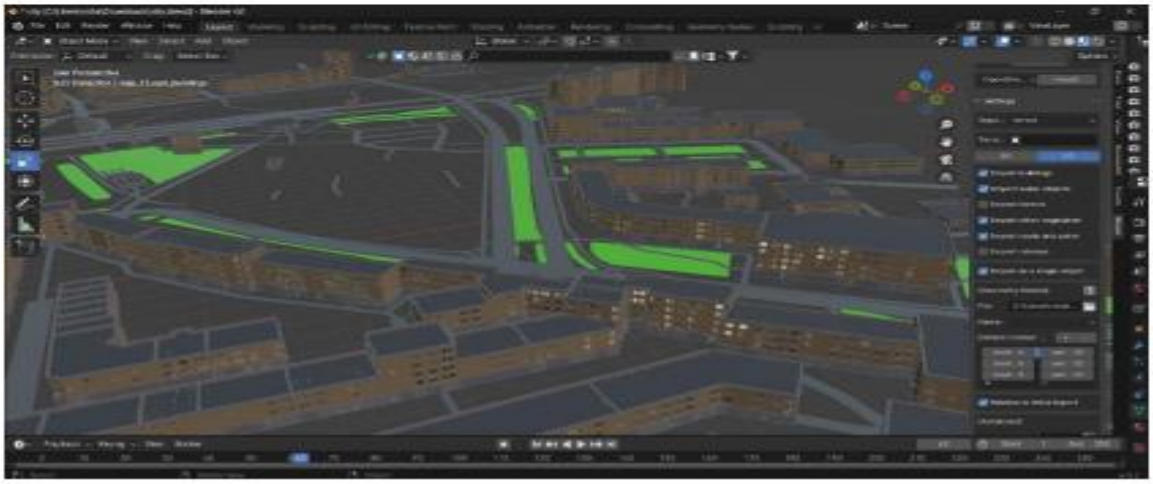
Collecting:

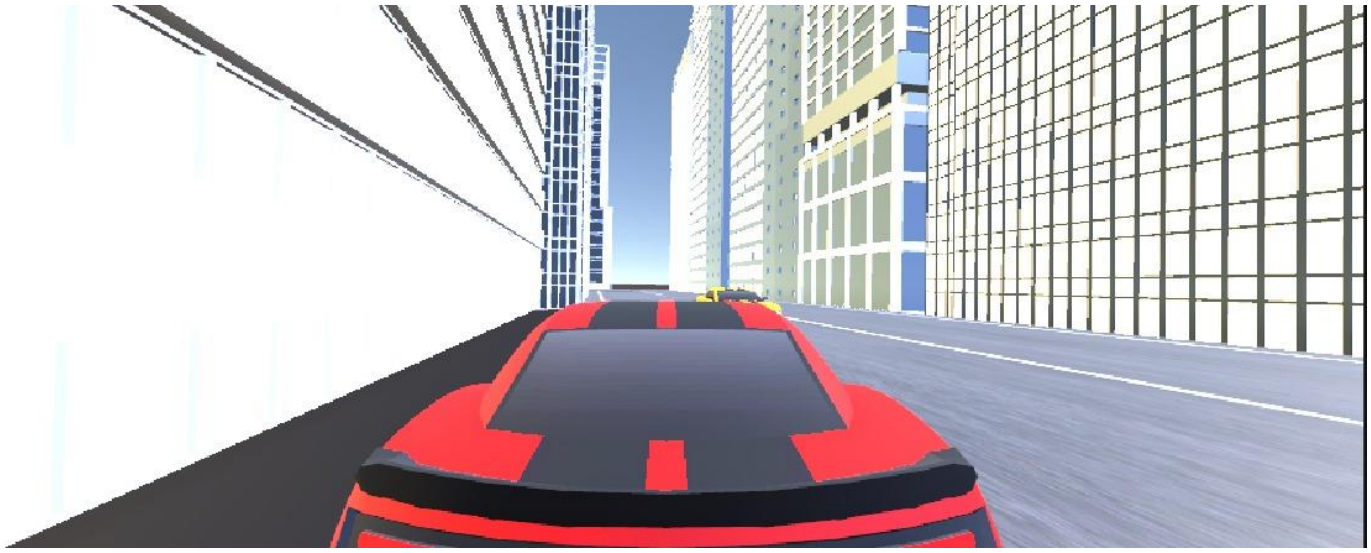
```
using System.Collections;
using System.Collections.Generic;
using UnityEditor.Search;
```

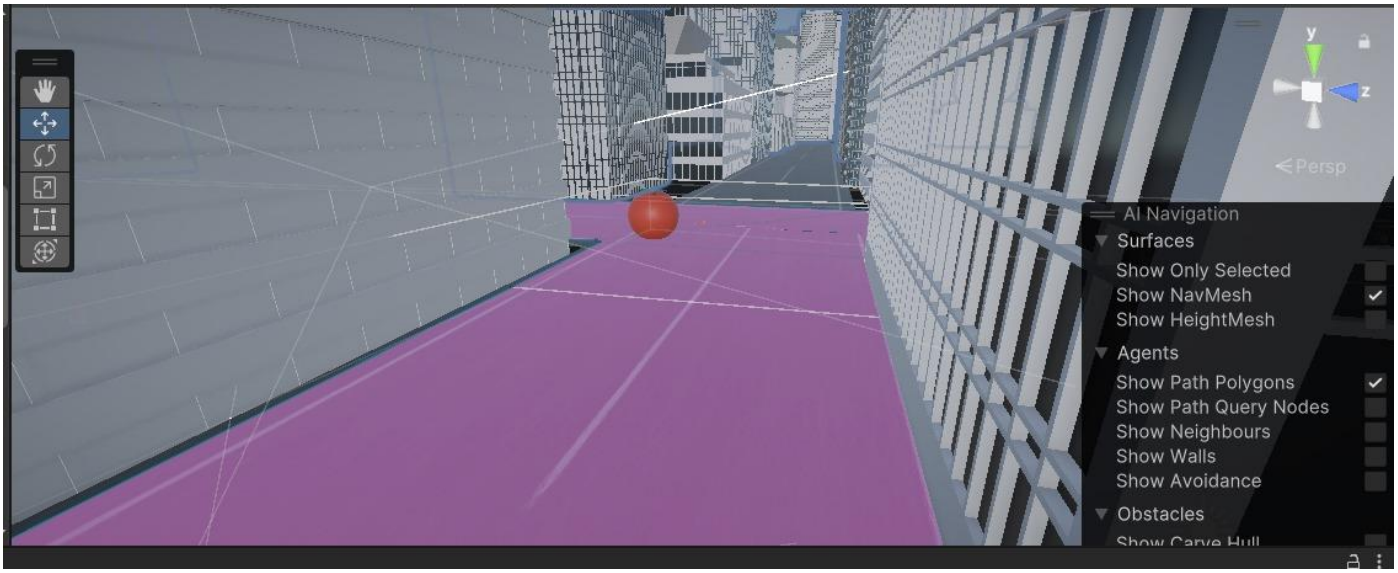
```
using UnityEngine;

public class Collecting : MonoBehaviour
{
    public List<string> Collected_items;
    void Start()
    {
        Collected_items = new List<string>();
    }

    // Update is called once per frame
    void Update()
    {

    }

    private void OnTriggerEnter(Collider other) {

        if(other.CompareTag("Collect"))
        {

            string itemType =
other.gameObject.GetComponent<CollectableComponent>().itemType;
            print("You Have collected: "+ itemType);
            Collected_items.Add(itemType);
            Destroy(other.gameObject);
        }
    }




}
```

Output (images):








y
z
Persp
AI Navigation
Surfaces
Show Only Selected
Show NavMesh
Show HeightMesh
Agents
Show Path Polygons
Show Path Query Nodes
Show Neighbours
Show Walls
Show Avoidance
Obstacles

Conclusion:

This project has explored the development of VR Thread City, an innovative approach to learning thread management concepts in operating systems. By utilizing virtual reality (VR) and a city metaphor, the project aims to address the challenges students often face in grasping these abstract concepts.

Overall, VR Thread City has the potential to revolutionize the way students learn thread management in operating systems. By leveraging the power of VR and interactive learning, this project can make thread management concepts more accessible, engaging, and ultimately, more understandable for students.