NIPES

Advances in Engineering Design Technology

Journal homepage: www.nipesjournals.org.ng

Design and Simulation of a Test Environment for Autonomous Underwater Vehicles

Ogodo O.E[a] , Osa E.[b]

[a]*Department of Mathematics and Computer Science, College of Natural and Applied Sciences, Western Delta University, Oghara, Delta State, Nigeria.*
[b]*Department of Electrical/Electronic Engineering, Faculty of Engineering, University of Benin, P.M.B. 1154, Benin City, Nigeria.*

ARTICLE INFO

Keywords:
Autonomous, Underwater, Game Engine, Unity 3D.

Received 28 Oct. 2021
Revised 22 November 2021
Accepted 25 November 2021
Available online 16 December 2021

https://doi.org/10.37933/nipes.a/3.2.2021.9

https://nipesjournals.org.ng
© 2021 NIPES Pub. All rights reserved

ABSTRACT

Game Engines are modern off-the-shelf, sophisticated simulation software that have proven beneficial when developing simulation testbeds for intelligent control of UVs. They readily provide the required physics, modeling, and rendering capabilities thereby reducing vehicle development time. In this paper, Unity 3D Game Engine is used to design and simulate an underwater environment for testing navigation of underwater vehicles. Various underwater scenarios were developed to test the functionality of the underwater environment by creating C# scripts and utilizing the internal physics properties of the Unity software. A virtual autonomous underwater vehicle was designed and deployed into the simulated environment to test the performance of the environment. Patrol scripts as well as obstacle avoidance scripts were written in C# programming language to condition the underwater vehicle in the simulated environment. The results showed that the underwater environment can test for navigation and patrol of Autonomous Underwater Vehicles.

1. Introduction

Unmanned Underwater Vehicles (UUVs) are all types of underwater robots which are operated with minimum or without the intervention of a human operator. This phrase is used to describe both Remotely Operated Vehicles (ROVs) and Autonomous Underwater Vehicles (AUVs). Remotely operated vehicles are teleoperated robots that are deployed primarily for underwater installation, inspection and repair tasks [1]. They have been used extensively in offshore oil industries due to their advantages over human divers in terms of higher safety, greater depths, longer endurance and less demand for support equipment [1]. During operation, the ROV receives instruction from an operator onboard a surface ship or other mooring platform through a tethered cable or acoustic link. AUVs on the other hand operate without the need of constant monitoring and supervision from a human operator. As such these vehicles do not have the limiting factor in its operation range from the umbilical cable typically associated with the ROVs. This enables AUVs to be used for certain types of missions such as long range oceanographic data collection where the use of ROVs is deemed impractical. They also find application in military purposes such as mine countermeasures, antisubmarine warfare and payload delivery [2]. Autonomous underwater vehicles (AUVs) are unmanned, self-propelled vehicles that are typically deployed from a surface vessel (shown in Figure 1) and can operate independently of that vessel for periods of few hours to several day [3]. These vehicles are required to execute different types of missions without the interaction of human

operators while performing well under a variety of load conditions and with unknown sea currents [4]. AUVs are typically categorized as either “cruising” or “hovering” vehicles depending on the shape and manoeuvrability of the vehicle, with the latter having high manoeuverability as a result of having several propellers [5]. These vehicles can also be equipped with a range of sensors such as chemical sensors, photo cameras, sonars, magnetometers, and gravimeters [5].

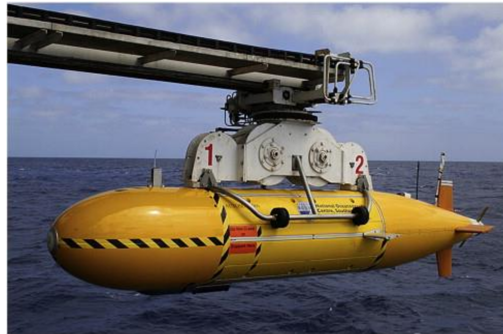

Figure 1: Autonomous Unmanned Vehicle (Source: The UK Natural Environment Council (NERC) Autosub6000 AUV [3].

Recent times have marked an increase in sub-sea exploration and marine research and thus necessitates development of autonomous underwater vehicles with particular emphasis on autonomy, navigation, object detection, energy sources and information systems [6]. Testing pools are usually not feasible due to space, cost and maintenance requirements. Water bodies such as lakes/ponds are usually remote and provide little control over the conditions. Conducting experiments and evaluations on a real physical system is not economically viable due to the possibility of physical strain and damage to the vehicle, high costs associated with manufacturing among other factors. Simulation modelling is the ideal solution in these circumstances especially in developing countries like Nigeria. This involves development of a virtual prototype of a physical system and analyzing its performance in the real world. Game Engines are numerous off-the-shelf platforms that have capable physics engines, rendering, and modeling facilities. As the name implies, these software systems are created for the design and development of modern 3D video games. Examples include Unity [7], Torque3D [8], Polycode [9] and CryEngine3 [10]. These range from well-maintained rendering and physics libraries to full toolkits that feature Graphic User Interface (GUI)-driven game world and object creation editors. Using these, a game designer can concentrate on the game itself and leave the physics and rendering to the engine. This concept can be applied to the test of navigation and patrol functions of underwater vehicles.

1.2 Overview of Underwater Simulators

There have been many simulators developed over the years for AUV development, including those facilitating intelligent control development. An early example was the DIS-Java-VRML simulator from the Naval Postgraduate School, which had sophisticated physics and rendering capabilities, making use of hardware help from Silicon Graphics workstations. It was one of the first to use both standardized communication (Distributed Interactive Simulation [DIS] protocol [11] and a standardized modeling language (Virtual Reality Markup Language VRML) [12]. While it was a very advanced simulation for its time, it also was written entirely from the ground up and it relied on specialized hardware, and its age clearly shows, at least visually. Its descendant, the AUV Workbench [13] while more modern, also required substantial resources for maintenance and continuous modernization. Over the years more beneficial simulators began to evolve. The use of

game engines for traffic and land vehicle simulation was carried out by [14]. [15] worked on GEAS. GEAS is a simulator for research in intelligent control of autonomous agents, especially AUVs. GEAS focuses on simulation at the level appropriate for high-level mission control software. It runs on a server machine, which may or may not be the same machine as the AUV controller (but usually will not be). Unity supports TCP sockets, and so GEAS can communicate with a server process that listens to a port for incoming connections. Authors in [16] presented URSim (Unity Ros Simulator), an open-source hybrid underwater simulator based on modular software framework - ROS (Robot Operating System) and a cross platform real-time game engine- Unity3D. It is an improvement on the above-named Simulators. The simulation described in this paper employs the URSim open-source framework as a background. However, it is unique in that it replaces the default capsule in URSim with a custom built Autonomous Underwater Vehicle along with a custom patrol C# script for navigation. The environment itself comprises four custom underwater scenes.

1.3 Unity 3D Game Engine

Any coherent simulation system can be developed using a graphics rendering mechanism for creating a virtual environment with realistic 3D models and a computational back end system where the corresponding data is processed and respective commands are issued. Unity 3D, a game engine used to develop half of the world's games was the first choice for the virtual environment development. It acts as the face of the whole system, providing a platform to create a visually realistic environment where the UUV's algorithms and control structures can be tested and experimented with. Unity 3D [17] is a cross-platform game engine primarily used to create games. Its powerful graphics rendering, physics engines and intuitive development tools make it the preferred choice for developing realistic, efficient and easily deployable simulations. Unity provides a robust physics engine to simulate real world physics in the form of rigid body kinematics, fluid dynamics and collisions. The Unity Asset Store with wide variety of ready-made plugins and assets that can be imported to accelerate development. Unity is backed by a large, active community of developers that has been growing since 2005, providing support and making it more approachable for new developers. Games and simulations developed in Unity can be deployed on most platforms including Windows, MacOS and Android. The Unity engine supports C# (pronounced "C sharp") as the programming language and provides several APIs to simulate real world physics. The development environment provided by Unity and its intuitive work-flow makes it easier for developers to get started with developing games and simulations. Drag and drop mechanisms make it easy to import and position objects, add components and attach scripts. The aesthetic specifications of game environments can be controlled using materials, shaders and textures in Unity. Ready-made environment packages and a vast asset store aid the development process. Overall Unity provides ease and flexibility to design user virtual environments which can mirror a variety of realistic scenarios.

2. Methodology

The methodology adopted for modelling the simulation environment is broken into three steps namely: Underwater 3D Environment modelling, Vehicle Design and Modelling, Hydrodynamic forces and collisions.

2.1. Underwater 3D Environment

The underwater environment of the simulation system has been developed keeping in mind the various capabilities of Underwater Vehicles in different underwater missions. There are four major sandboxing environments that have been created in this standing simulation. Each sandbox (test-

beds) is unique in its own way. In order to create the underwater environment certain properties had to be configured appropriately namely: Main Camera, Right Perspective Camera, Non-Player Character, Floor, Directional Light, Top Camera, Locator Sphere, Pointers Patrol, Environmet, ColliderLevels, Water, Projectors, CaveLevelWalls.
Scripts for various configuration in C# programming language are contained in the Appendix Section.

2.1.1 Sandbox One

The environment developed as shown in Figure 1 acts a virtual test-bed for wholesome evaluation of the vehicle's capabilities and performance. It includes the simple horizontal terrain aimed at training the robot to execute a simple point-to-point navigation without patrol functions. Rocks and clear sea bed paths have been placed for obstacle avoidance training, a piping network for leakage detection and a geo-relative game object (Locator Sphere) with clear shaders has also been placed for target acquisition related tasks respectively.

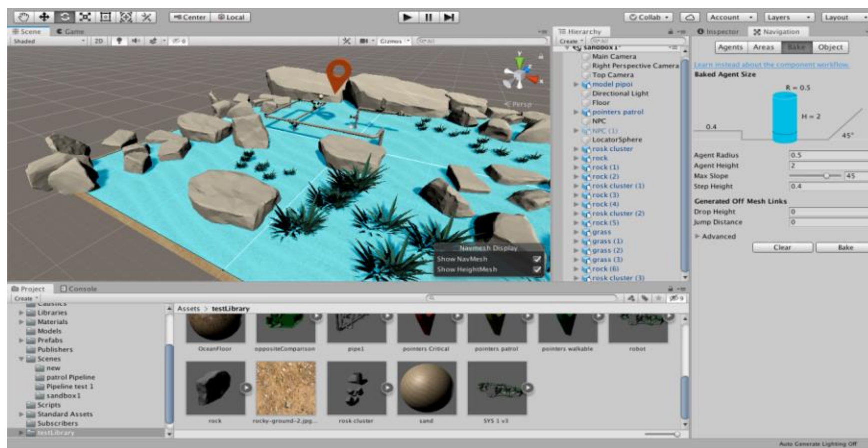

Figure 1: Sandbox one, simple point-to-point voyage

2.1.2 Sandbox Two

Unlike the previous sandbox, this one has a different pipping system with an attached patrol script (shown in appendix) for training in terms of leakage checks across serval points in the pipping system. The pipping network covers a much larger sea bed frame and wider placed seabed rocks for a more patrol-friendly environment.

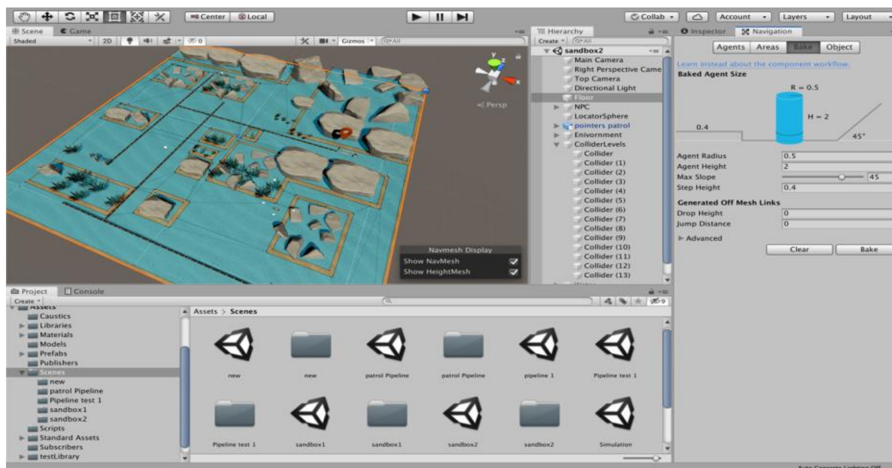

Figure 2: Sandbox two, larger piping network and patrol

2.1.3 Sandbox Three

This testbed as shown in Figures 3a and 3b carries more advanced navigational approach to design. Its centers on maneuvering capability training for the underwater vehicle. Red and green hoops (circular gates) have been sunk into the ground for a training session where by the UV will autonomously navigate through green hoops and evades red loops during voyage. This test simply points to the test for best voyage/navigation algorithm, to which collision avoidance and mapping training can be ascertained. Modulation and recurrent retaining of the UV via the navigation bake function provided by Unity3D enables the use of Artificial intelligence algorithms to surf through explicit options and concurrently helps in sensor functions whilst deciding which hoop is acceptable for navigation.

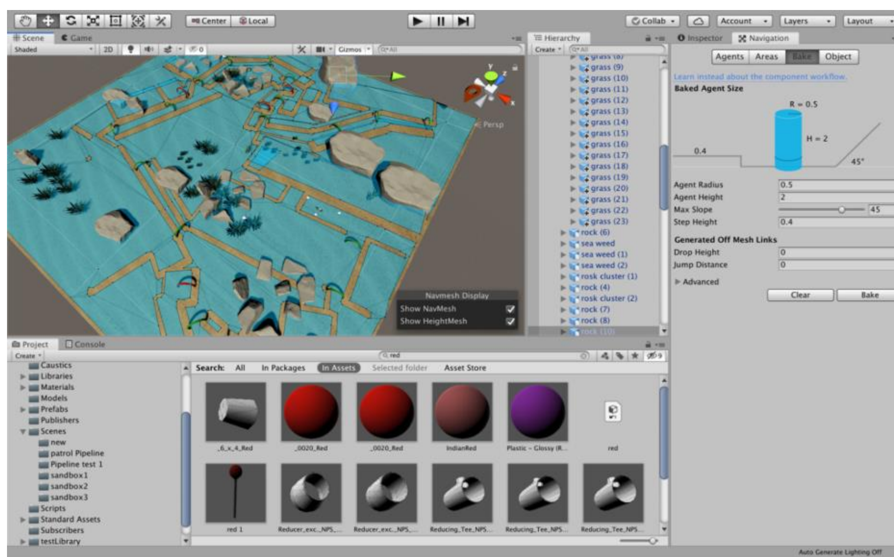

Figure 3a: Sandbox three, orthographic view of green-red hoop testbed

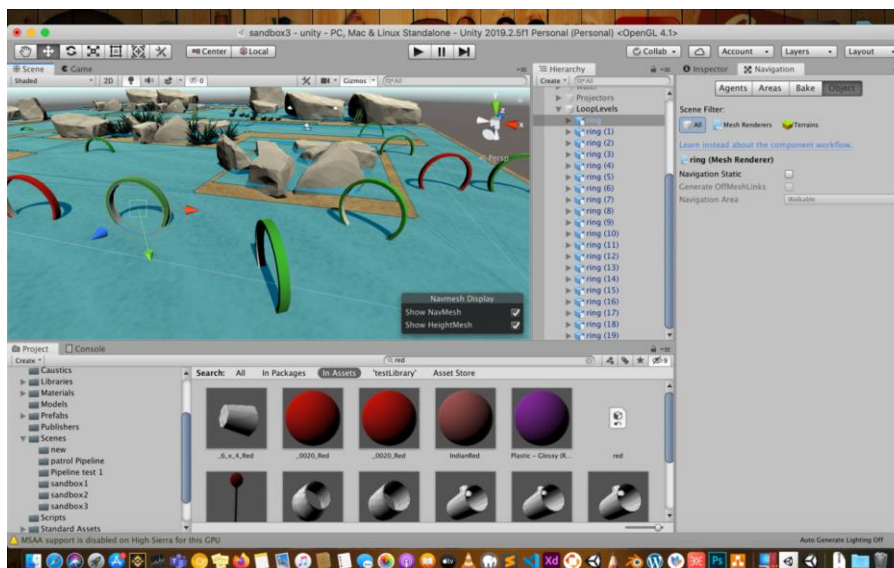

Figure 3b: Sandbox three, perspective view of Colour-difference hoop

2.1.4 Sandbox Four

As a hectic task, this sandbox was driven with complexity. It features a 3-level navigational course. It is primarily an underwater cave system that houses three lateral layers of navigable rock paths.

Unlike, the other sandboxes; this fourth testbed was created to prove different depth navigation for an autonomous underwater vehicle.

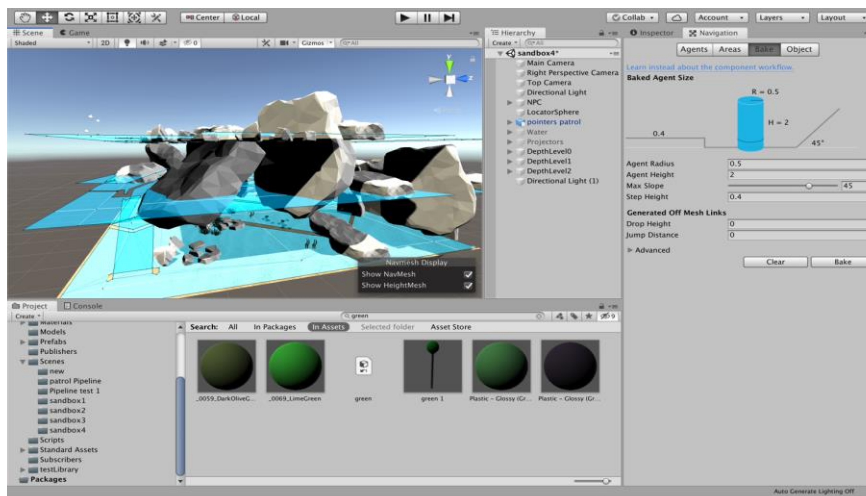

Figure 4: Sandbox four, Underwater Cave System.

2.2 Vehicle Design and Modelling

2.2.1 Vehicle Design

The vehicle design was carried out using Fusion 360 Software. The approach encompassed certain objectives to fulfill a simple system for the vehicle design and mission requirements which included:

1. To select suitable component parts from already existing technology.

2. To ensure the weight of the vehicle and propulsion parts relative to power and energy kept low.

3. Ease of a control system for the vehicle.

4. A high efficiency of the control and propulsion system of the vehicle in operation.

5. To design for a high degree of freedom (mobility) of the vehicle in water.

6. The choice of propulsion system to run noiselessly in operation.

7. Should the propulsion system fail, the vehicle should be positively buoyant so it can rise to the surface for retrieval.

To better understand the performance characteristics of the design the approach here reduces the engineered system to a form of an equivalent spheroid (a prolate spheroid, neutrally buoyant with the same length and mass as our AUV design). For this assumption, the following holds true

$$D_s = \sqrt{\frac{6m}{\rho \pi L}} \qquad (1)$$

Where

D_s =Equivalent model diameter(m)

m =Mass(kg)

ρ =Water density(kg/m^3)

L =Length(m)

For the AUV in water at an assumed temperature of 20℃, its

$m = 1.75kg$

$L = 0.3154m$

At 20℃

$\rho = 998.23kg/m^3$

$\nu = 1.002\times10^{-6}m^2/s$

$$D_s = \sqrt{\frac{6\times1.76}{998.23\times\pi\times0.3154}}$$

$D_s = 0.1033m$

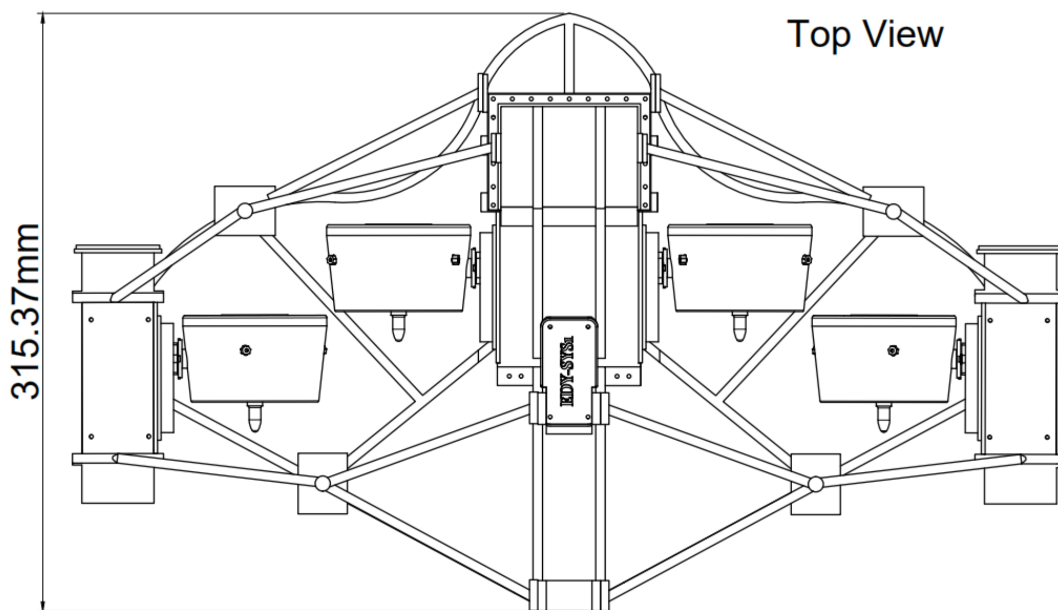

Figure 5: Length of the AUV in mm from the CAD workspace

By defining the equivalent spheroid by its mass and slenderness ratio $\frac{L}{D_s}$ For 1.1> $\frac{L}{D_s}$ <15 an error of less than 1% can be approximated for the wet surface area.

$$\frac{L}{D_s} = \frac{0.3154}{0.1033} = 3.0532$$

$$A_s \approx dm^{\frac{2}{3}} \quad \text{... (2)}$$

Where

A_s =Wet surface area of equivalent model(m^2)

d =A constant for geometrically similar spheroids operating in the same fluid

$$d = \left(\frac{1}{\rho}\right)^{2/3}\left(-0.0122\left(\frac{L}{D_s}\right)^2 + 0.5196\frac{L}{D_s} + 4.2732\right) \quad \text{.......................................(3)}$$

$$d = \left(\frac{1}{998.23}\right)^{2/3}\left(-0.0122(3.0532)^2 + 0.5196(3.0532) + 4.2732\right)$$

$$d = 0.0574$$

$$A_s = 0.0574 \times 1.76^{\frac{2}{3}}$$

$$A_s = 0.0837 m^2$$

At this point, it is suitable to assume a low running speed for the AUV operation in order to calculate the ideal drag this equivalent spheroid would face. While the following holds true, they are but a guide to understand our AUV operation in water.

With $U = 0.1 m/s$

The International Tow Tank Conference plot of frictional resistance cf versus Reynolds number gives the following relationship

$$Cf = \frac{0.075}{[(\log \mathrm{Re}) - 2]^2} \quad \text{..(4)}$$

$$\mathrm{Re} = \frac{VL}{\nu} \quad \text{... (5)}$$

Where

Cf =A Skin friction number

Re= Reynolds number

L= Vehicle length

ν = kinematic viscosity of water

$$\mathrm{Re} = \frac{0.1 \times 0.3154}{1.002 \times 10^{-6}}$$

Re=31477.0459

$$Cf = \frac{0.075}{[(\log 31477.0459) - 2]^2}$$

$$Cf = 0.0120$$

For simplicity, consideration will be on the viscous drag experienced by the submerged AUV and thus,

$$C_D = C_v = C_f(1 + k) \ldots\ldots\ldots\ldots\ldots (6)$$

For which the form factor (1+k) predicted empirically can be estimated by

$$(1+k) = 1 + 1.5\left(\frac{L}{D_s}\right)^{-3/2} + 7\left(\frac{L}{D_s}\right)^{-3} \ldots\ldots\ldots\ldots\ldots (7)$$

Where

C_D =Drag coefficient of the towed system

$$(1+k) = 1 + 1.5(3.0532)^{-3/2} + 7(3.0532)^{-3}$$

$$(1+k) = 1.5271$$

$$C_D = 0.0120(1.5271)$$

$$C_D = 0.0183\,.$$

Testing the designed model is very essential to avoid wrong building and misbehavior during voyage. The modelling task is divided into two categories namely Kinematics and Dynamics. Fig. 6 shows a six degrees of freedom (6DoF) model of the vehicle. The vehicle is modelled to control heave (motion along z-axis), roll (rotation about x-axis), pitch (rotation about y-axis), yaw (rotation about z-axis), surge (motion along x-axis) and sway (motion along y-axis).

2.2.2 Dynamics

We further divide the dynamics of the vehicle based on two possible motions - Translational Motion and Rotational Motion. Unity physics engine account for several forces like hydrostatic, lift, drag, thrust, external forces acting on the vehicle simulating the motion.

2.2.3 Kinematics

To demonstrate the kinematics, two reference frames were added, body-fixed frame and the inertial frame. The body-fixed frame moves relative to inertial frame, hence simulating the linear and

angular velocities. The position and orientation are described in reference to inertial frame thus simulating the degree of freedom of vehicle.

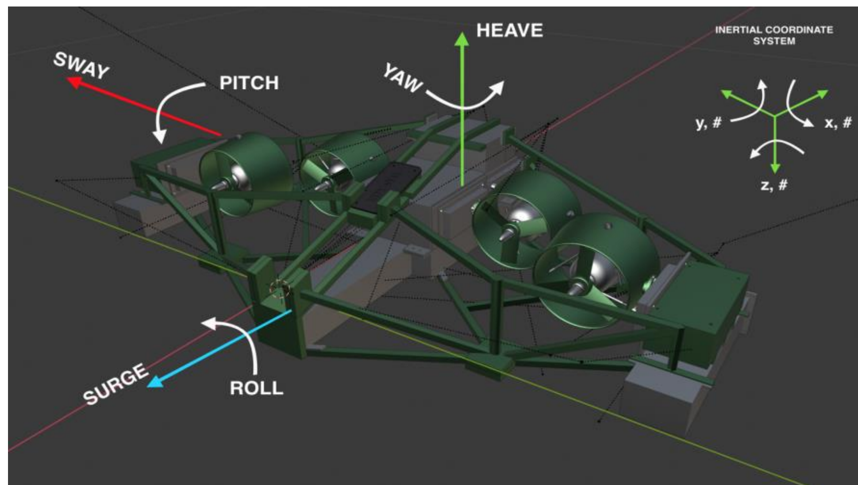

Figure 2.6: Vehicle Modelling.

2.3 Hydrodynamic Forces and Collisions

Scenes in Unity are created using 3D or polygonal meshes. Meshes are the foundational structures of all games made in Unity. All objects in a scene are made up of meshes. Each mesh comprises of vertices and edges in 3D space that come together to form multiple triangles. Therefore, the surface of each Game Object can be considered to be made up of multiple small triangles.

Simulation of any force on an object requires application of force on the meshes' triangles. The proposed simulation system accounts for the various forces like buoyancy and drag that act on a solid body immersed underwater.

2.3.1 Buoyancy and drag

To enable a Game Object to experience physics-based forces, Unity provides Rigid body plugin. It allows the object to experience mass, velocity, gravity and drag. It can also be used to impose constraints on linear and angular movements and to detect collisions. To simulate buoyancy, corresponding C# scripts are attached to the Game Object to apply an upward force to the centers of all the submerged triangular faces of the vehicle's mesh.

$$B = \rho g v \text{ -------------------------- (8)}$$

$$\text{Where, } v = zS \text{--------------------------- (9)}$$

In (8), the buoyant force B, can be calculated using the Archimedes principle while ρ represents the density of the liquid, g represents the gravitational acceleration and v represents the volume of the water above an individual triangle in the mesh. As demonstrated in (1) this volume can be calculated from (9) using the product of the surface area of the triangle, S, and the distance between the center

of the triangle and the surface of the water, z. If this upward force is greater than the gravitational force, then the vehicle floats on the surface of the water plane. Simulation of the damping forces, drag and angular drag coefficients were specified using the Rigid body interface inside Unity platform.

3. Results and Discussion

The test of each scene as described in the methodology was carried out by trying out the navigation of a virtual Autonomous Vehicle in the respective scenes of the underwater simulation environment. This is shown in Figures 7 to 10. The Underwater Vehicle has an aerial and directional lamp added to it, to ensure proper navigation, scanning, visibility during voyage in terms of camera feedback and picture taking.

Table 1: Parameters for Baked Test Agent in Unity

PARAMETER	VALUES
Agent Height	2
Agent Radius	0.5
Maximum Slope	45
Step Height	0.4

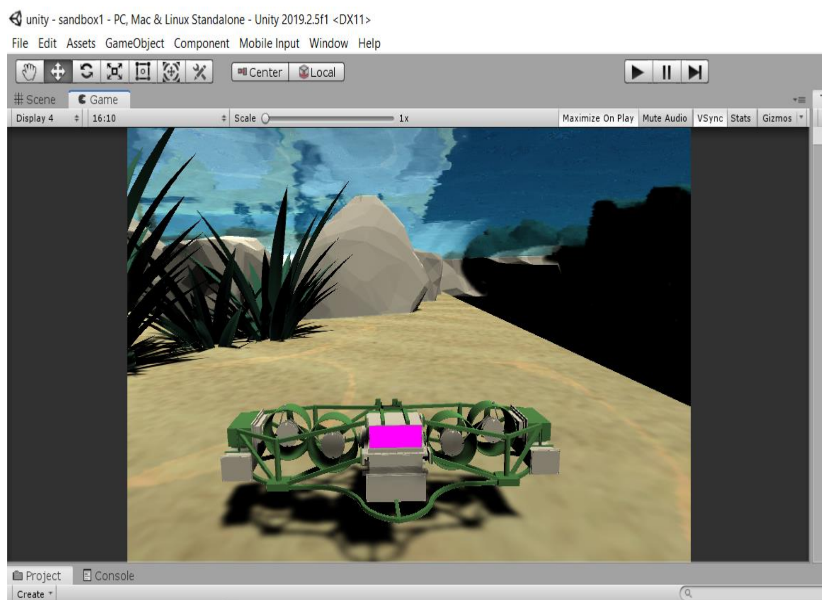

Figure 7: Test Case for Sandbox One

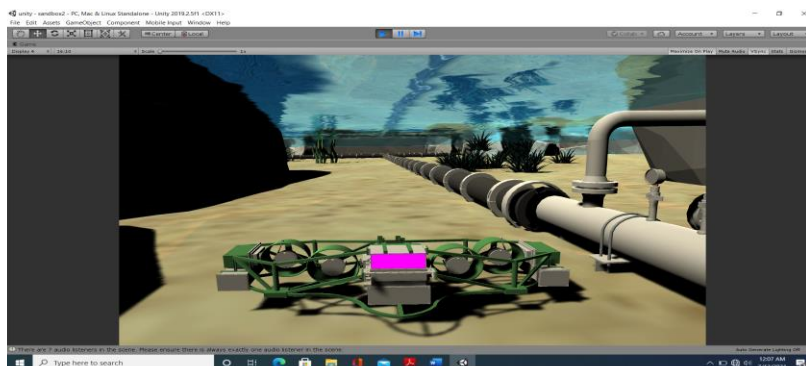

Figure 8: Test Case for Sandbox Two

Figure 9: Test Case for Sandbox Three

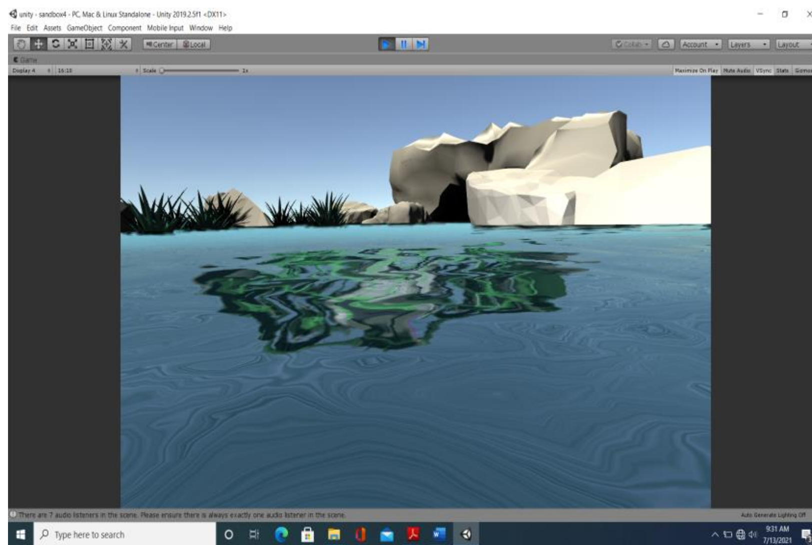

Figure 10a: Test Case for Sandbox Four Water Surface Level

Figure 10b: Test Case for Sandbox Four Underwater Cave Level

3.2 Discussion

Figure 7 shows that the vehicle could perform simple navigation functions in the simulated environment which is the objective of Sandbox one. Figure 8 shows that the vehicle could perform patrol functions in the simulated environment of sandbox two. The vehicle can be seen patrolling an underwater piping network. Figure 9 shows that the UV can autonomously navigate through green hoops and evades red loops during voyage. This test simply points to the test for best voyage/navigation algorithm, to which collision avoidance and mapping training can be ascertained. Modulation and recurrent retraining of the UV via the navigation bake function provided by Unity3D enables the use of Artificial intelligence algorithms to surf through explicit options and concurrently helps in sensor functions whilst deciding which hoop is acceptable for navigation. In this scenario, the A* navigation algorithm has been deployed. As shown in Figures 10a and 10b, Sandbox four proves different depth navigation for an autonomous underwater vehicle. It navigates the water surface and then descends below to lower cave levels. These outcomes thus validate the C# codes (shown in Appendix) implemented.

4. Conclusion

This paper describes the simulation of an underwater environment. Such environments serve as low-cost arenas for experimenting underwater vehicles to ascertain their functionalities before actual fabrication. Various tests prove that the simulation environment can serve for testing virtual underwater vehicles in terms of navigation, patrol, obstacle avoidance and other functions. It is recommended that the academia namely universities and other tertiary centres of learning adopt more of these procedures of employing simulation environments to test systems and machines before actual deployment. The huge advantage of low cost cannot be overemphasized especially in environments where it is seemingly difficult to get adequate equipment for carrying out experiments.

References

[1] A. Budiyono (2009) Advances in Unmanned Underwater Vehicles Technologies; Modelling, Control and Guidance Perspectives. Indian Journal of Marine Sciences. Vol 38. (3), pp. 284-295.

[2] U.S Department of the Navy (2004), The Navy Unmanned Undersea Vehicle (UUV) Master Plan, 9 Nov.2004.

[3] R.B. Wynn, V.A.I. Huvenne, T.P. Le Bas, B.J. Murton, D.P. Connelly, B.J. Bett, H.A. Ruhl, K.J. Morris, J. Peakall, D.R. Parsons, E.J. Sumner, S.E. Darby, R.M. Dorrell, J.E. Hunt (2014), "Autonomous Underwater Vehicles (AUVs): Their Past, Present and Future Contributions to The Advancement of Marine Geoscience," Marine Geology, vol. 352, pp. 451-468.

[4] T. Elmokadem, M. Zribi and K. Youcef-Toumi (2016), "Trajectory tracking sliding mode control of underactuated AUVs," Nonlinear Dynamics, vol. 84, no. 2, pp. 1079-1091.

[5] V.A. Huvenne, K. Robert, L. Marsh, C.L. Iacono, T. Le Bas and R.B. Wynn (2018), "ROVs and AUVs," in Submarine Geomorphology: Springer, pp. 93-108.

[6] Contest.techbriefs.com, Autonomous Submarine Vehicle- Create the Future Design Contesthttps://contest.techbriefs.com/2016/entries/machinery-automation-robotics/6918.

[7] W. Goldstone (2009). Unity Game Development Essentials. Packt Publishing Ltd.

[8] J. Lloyd (2004). The torque game engine. Game Development Magazine.

[9] I. Safrin (2013). Polycode web site. http://polycode.org, accessed 3/28/2013.

[10] H. Seeley (2007). Game technology 2007: Cryengine2. In ACM SIGGRAPH 2007 computer animation festival, page 64. ACM.

[11] D.A Fullford (1996). Distributed interactive simulation: its past, present, and future. In Proceedings of the 28th conference on Winter simulation, pages 179{185. IEEE Computer Society.

[12] J. Hartman and J. Wernecke (1996). The VRML 2.0 handbook: building moving worlds on the web. Addison Wesley Longman Publishing Co., Inc.

[13] D.T Davis and D. Brutzman (2005). The autonomous unmanned vehicle workbench: mission planning, mission rehearsal, and mission replay tool for physics-based x3d visualization. In Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology.

[14] J.L Pereira and R.J Rossetti (2012). An integrated architecture for autonomous vehicles simulation. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, pages 286{292. ACM.
[15] A.B Strout and R.M Turner (2013). A Game Engine-Based Simulator for Autonomous Underwater Vehicles. In the Proceedings of the 18th International Symposium on Unmanned Untethered Submersible Technology (UUST), Portsmouth, NH, August, 2013.
[16] Katara, P., Khanna, M., Nagar, H. and Panaiyappan, A. (2019) Open Source Simulator for Unmanned Underwater Vehicles using ROS and Unity3D.
[17] Unity3d", [online] Available: http://unity3d.com.

Appendix

AUV PATROL SCRIPT IN C#

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
//AI patrol for AUV simulation.

public class NPCSimplePatrol : MonoBehaviour
{
   //Dictates whether the agent waits on each node
   [SerializeField]
   bool _patrolWaiting;

   //The total time we wait at each node
   [SerializeField]
   float _totalWaitTime = 3f;

   //The probability of switching direction.
   [SerializeField]
   float _switchProbability = 0.2f;

   //The list of all patrol nodes to visit.
   [SerializeField]
   List<Waypoint> _patrolPoints;

   //Private variables for base behaviour.
   [SerializeField]
   NavMeshAgent _navMeshAgent;
   int _currentPatrolIndex;
   bool _travelling;
   bool _waiting;
   bool _patrolForward;
   float _waitTimer;

   // Use this for initialization
   public void Start()
   {
      _navMeshAgent = this. GetComponent<NavMeshAgent> ();

      if (_navMeshAgent == null)
      {
         Debug.LogError("The nav mesh agent component is not attached to " + gameObject.name);
      }
      else
      {
         if(_patrolPoints != null && _patrolPoints.Count >= 2)
         {
            _currentPatrolIndex = 0;
```

```
            SetDestination();
        }
        else
        {
            Debug.Log("Insufficient patrol points for basic patrolling behaviour.");
        }

    }
}

public void Update()
{
    //Check if Robot is close to the destination.
    if(_travelling && _navMeshAgent.remainingDistance <= 1.0f)
    {
        _travelling = false;

        //if the robot is goint to wait, then wait.
        if(_patrolWaiting)
        {
            _waiting = true;
            _waitTimer = 0f;
        }
        else
        {
            changePatrolPoint ();
            SetDestination();
        }
    }

    //Instead if we're waiting.
    if(_waiting)
    {
        _waitTimer += Time.deltaTime;
        if (_waitTimer >= _totalWaitTime)
        {
            _waiting = false;

            changePatrolPoint();
            SetDestination();
        }
    }
}

private void SetDestination()
{
    if (_patrolPoints != null)
    {
        Vector3 targetvector = _patrolPoints[_currentPatrolIndex].transform.position;
        _navMeshAgent.SetDestination(targetvector);
        _travelling = true;
    }
}

// <summary>
// Selects a new patrol point in the available list, but
// also with a small probability allows the robot to move forward or backwards.
// </summary>
private void changePatrolPoint()
{
```

```
        if (UnityEngine.Random.Range(0f, 1f) <= _switchProbability)
        {
            _patrolForward = !_patrolForward;
        }

        if (_patrolForward)
        {
            /**
            _currentPatrolIndex++;

            if(_currentPatrolIndex >= _patrolPoints.Count)
            {
                _currentPatrolIndex = 0;
            }
            */

            _currentPatrolIndex = (_currentPatrolIndex + 1) % _patrolPoints.Count;
        }
        else
        {
            /**
            _currentPatrolIndex--;

            if(_currentPatrolIndex < 0)
            {
                _currentPatrolIndex = _patrolpoints.Count - 1;
            }
            */
            if (--_currentPatrolIndex < 0)
            {
                _currentPatrolIndex = _patrolPoints.Count - 1;
            }
        }
    }
}
```

UNDERWATER EFFECT SCRIPT IN C#

```
using UnityEngine;
using System.Collections;

public class underwaterEffect : MonoBehaviour
{
    public float waterHeight;

    private bool isUnderwater;
    private Color normalColor;
    private Color underwaterColor;

    // Use this for initialization
    void Start()
    {
        normalColor = new Color(0.5f, 0.5f, 0.5f, 0.5f);
        underwaterColor = new Color(0.22f, 0.65f, 0.77f, 0.5f);
    }

    // Update is called once per frame
    void Update(){
        if ((transform.position.y < waterHeight) != isUnderwater)
        {
```

```
        print(transform.position.y);
        print(waterHeight);
        isUnderwater = transform.position.y < waterHeight;
        if (isUnderwater) SetUnderwater();
        if (!isUnderwater) SetNormal();
      }
    }

    void SetNormal()
    {
      RenderSettings.fogColor = normalColor;
      RenderSettings.fogDensity = 0.01f;

    }

    void SetUnderwater()
    {
      RenderSettings.fogColor = underwaterColor;
      RenderSettings.fogDensity = 0.02f;

    }
}
```