1 INTRODUCTION

1.1 Description

A central aspect of progress for edutainment and computer gaming applications is their steady increase in the immersiveness of the user experience. Historically, this routinely sprang from ever more involved and complex plots as well as crisper and more spectacular 3-D graphics. Then, games combined the two resulting in blazing graphics on top of challenging interactive stories and other such things of personalization. Game control, however, remained quite crude; mostly relying on text input, mouse clicks, joy pads, and joysticks. Maybe this is why certain genres supporting much nicer control devices flourish so well. Car racing games, for instance, achieve an extremely immersive user experience by means of realistic steering wheels and gear shifts paired with real-time force feedback as their preferred means of game control. During roughly the past 5 years, a general shift towards better and more immersive game control started to take hold. Pioneered by the "EyeToy" and dancing mats for Sony game consoles, not only academics but also the entertainment industry realized that people's real-world physical actions need to directly affect their playing reality. Contrasting with others, we envision to employ inexpensive camera technology to achieve this – preferably coloured gloves worn by people. This provides inexpensive basis for motion analysis while letting users move and roam about freely, independent of any additional infrastructure. Hence, we deem them superior to and prefer them over other approaches pursuing similar goals. Current hardware developments seem to support our notion in this respect. Select gaming consoles (e.g., Nintendo DS and Wii, Microsoft Kinect) and other electronic gadgets already come equipped with integrated motion sensors.

1.2 Motivation

- The facets of games have saturated to PCs and consoles which have raised concern over gamer's health and physical fitness.
- Games and gaming equipment evolved with time but the way game is played remained same.
- Gamers skip their physical exercise routine and remain indoors just tapping the buttons of the controller.
- The learning curve of the system should be gentle, hence faster learning game interaction and handling
- Provide the most natural way to interact with the system which will make easy for gamers to understand

1.3 Problem Formulation & Methodology Used

The project is to implement a car simulation program with hand gestures as input to the program. These gestures are recorded by Microsoft Kinect device and processed by the Microsoft Kinect drivers and SDK. This processed data is later fed to the Kinect Unity Wrapper package that is a part of the Unity3d game engine. Microsoft Kinect Software Development Kit(SDK) is package of drivers and development libraries that allows the operating system to communicate with Kinect and also allows developers to use its features.

As seen in the figure 1.1, the data flow is from inner circle to outer. The Game unit displays the graphics on display unit, i.e. is the monitor. The user responds to visuals and accordingly to give the gesture input and the above loop continues till the exit condition is reached.

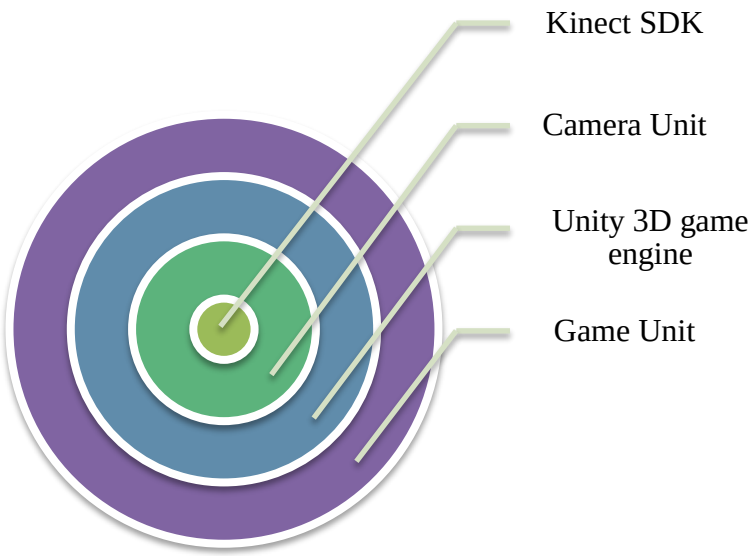

Fig 1.1: Project Formulation

1.4 Relevance of the project

The project is highly relevant in the real world scenario with further research applications as well. It provides newer ways of interacting with the system that have not been used earlier. The way to use and interact with the system is easily compatible with the way we interact in day to day life, like we sometimes tend to use gestures while talking, etc.

A normal car driver is familiar with controlling the car using the steering wheel. The project simulates those conditions hence providing a more natural way of interacting with the system. It does not constrain the user to sit in front of the system with keyboard and mouse but gives the freedom to interact with the system to without touching it.

1.5 Scope of the Project

- The project will only provide hand gesture input to the computer system.
- The user needs to maintain required distance from the system.
- The user will be able to control the aspects of the application by hand gestures only.
- The application will respond to specific gestures only.

1.6 Objectives of the Project

- Aim to provide new ways of using any computer system.
- Create visually appealing car simulation.
- Provide input to the system by using gestures only.

2. REVIEW OF LITERATURE

The Project is based on the patented technology of Microsoft Kinect, Unity3d Game Engine and MonoDevelop's Open Source framework based on .Net framework. It is written in C# and UnityScript(variant of JavaScript) with graphical models made in Autodesk's FBK format.

Microsoft Kinect is a motion sensing input device by Microsoft for the Xbox 360 video game console and Windows PCs. Based around a webcam-style add-on peripheral for the Xbox 360 console, it enables users to control and interact with the Xbox 360 without the need to touch a game controller, through a natural user interface using gestures and spoken commands.

Unity (also called Unity3D) is a cross-platform game engine with a built-in IDE developed by Unity Technologies. It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices, and is utilized by over one million developers. Unity is primarily used to create mobile and web games, but can also deploy games to consoles or the PC. The game engine was developed in C/C++, and is able to support code written in C# or JavaScript. It grew from an OS X supported game development tool in 2005 to the multi-platform game engine that it is today.

MonoDevelop is an open source integrated development environment for the Linux platform, Mac OS X, and Microsoft Windows, primarily targeted for the development of software that uses both the Mono and Microsoft .NET frameworks. MonoDevelop integrates features similar to those of NetBeans and Microsoft Visual Studio, such as automatic code completion, source control, a graphical user interface (GUI) and Web designer. MonoDevelop integrates a Gtk# GUI designer called Stetic, It currently has language support for C#, Java, Boo, Visual Basic.NET, Oxygene, CIL, Python, Vala, C and C++

FBX is a file format (.fbx) is used to provide interoperability between digital content creation applications.

3. SYSTEM STUDY AND ANALYSIS

3.1 Existing System

The existing systems consists of a:

- Typical desktop with CPU, monitor, keyboard or
- Laptop with similar components packaged into a single device or
- Touchscreen device.

These existing systems restricts the user to the physical proximity of the device that does not give much freedom to him/her.

Fig 2.1: A typical User-System Interaction.

3.2 Proposed System

The proposed system of the project removes the restrictions mentioned in [3.1] for the user by providing a larger volume for movement of the user. The user can be at a larger distance from the system which provides more freedom in movement. The proposed system has the CPU, monitor and Microsoft Kinect as input device. With help of Kinect, the possible ways to user interaction increases.

3.4 Requirement Analysis

3.4.1Overview of Requirements:

The 'Car Simulation using Kinect as input' needs to provide gameplay input with player's hand gestures. The project depicts real life environment simulation along with tracking of player's hands in real-time. The following is considered:

a. **Game:** To depict real life environment the project needs to have models of cars, structure, different tracks, terrains, etc. along with proper sound effect at any part of the playing region.
b. **Camera Unit:** System requires a camera unit to track the player's movement.
c. **Input Interface:** An interface must be required to take the camera output as the input and process those input to move an object on the screen in the system exactly as movement of the player's hand.

3.5 Requirement Specification

- System's compatibility with camera unit.
- Player within the camera's visible range.
- Single player detection.

3.6 Requirement Validation

- The models in the game must have texture in accordance with real life objects and scene.
- The models in the 3D environment of game should be in proportion with each other.
- The distance of player should not exceed below or above the sensing area of the camera unit.
- Only single user (Player) should be present in front of camera unit to play the game correctly.

3.6 Use-Case Diagrams and description

3.6.1 Use-Case Diagram

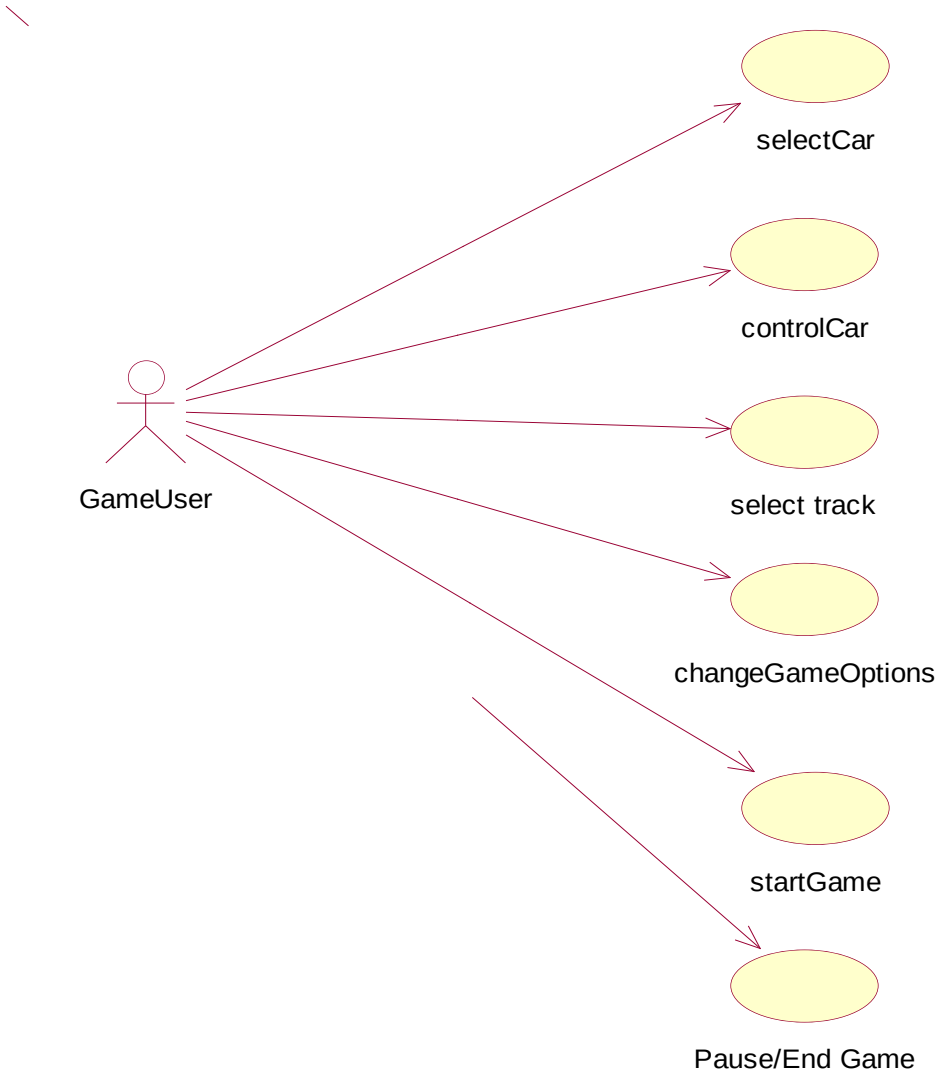

Fig 3.1: Use-Case Diagram

3.6.2 Use-Case Description

Use Case Name	Select Car
Actors	1.GameUser
Goal	1. To select a car to play a level
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.1: Select Car Use-Case

Use Case Name	controlCar
Actors	1.GameUser
Goal	1. To control the movements of car within the game
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.2: controlCar Use-Case

Use Case Name	Select Level
Actors	1.GameUser
Goal	1. To select a particular track/level to play
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.3: Select Level Use-Case

Use Case Name	changeGameOptions
Actors	1.GameUser
Goal	1. To change various settings within the game like sound, control calibration.
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.4: GameUser Use-Case

Use Case Name	Start Game
Actors	1.GameUser
Goal	1. To load all game components and start the game
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.5: Start Game Use-Case

Use Case Name	Pause/End Game
Actors	1.GameUser
Goal	1. To pause or completely exit from current game level play
Pre-Condition	1. Camera unit detects the GameUser's actions
Exceptions	1. Detection Error

Table 3.6: Pause/End Game Use-Case

4. ANALYSIS MODELING

4.1 Data Modeling

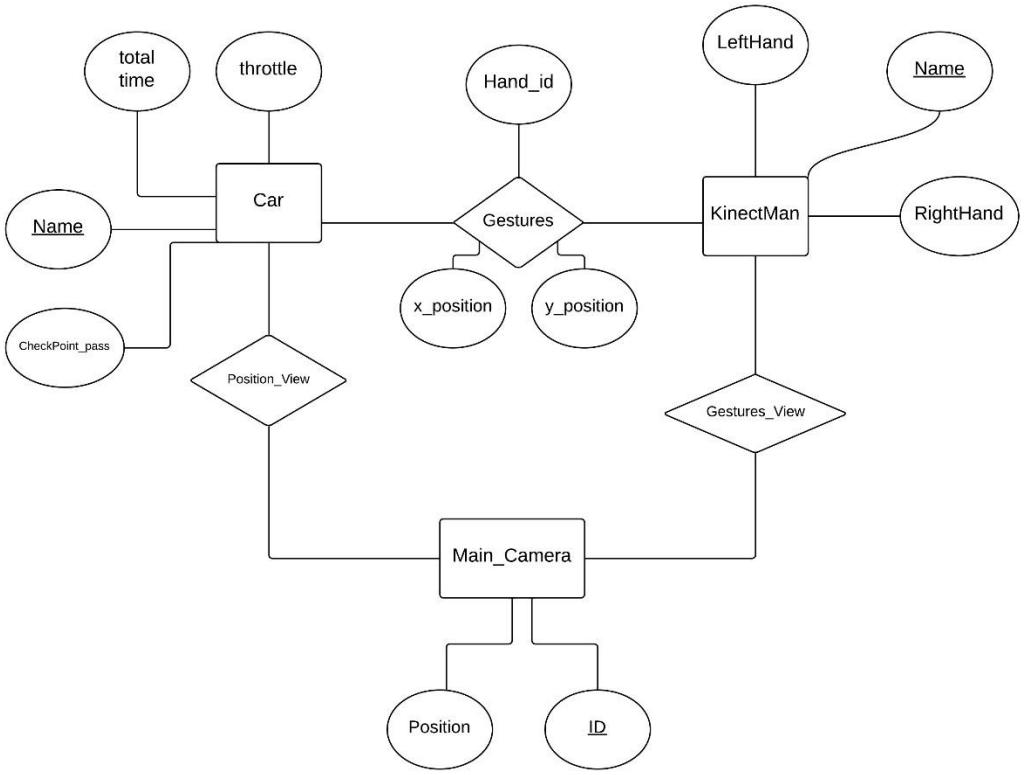

Fig 4.1: Entity Relationship Diagram

4.2 Data Dictionary

Game_Camera

Name	Data type	Constraints
CameraID	String	PRIMARY KEY
Image_stream	Blob	NOT NULL
ColurId	String	NOT NULL, UNIQUE
ColourStartX	Int	NOT NULL
ColourStartY	Int	NOT NULL
ColourArea	Int	NOT NULL

Table 4.1: Game_Camera Data

Car

Name	Data type	Constraints
Car_id	String	PRIMARY KEY
Model	String	NOT NULL
Category	String	NOT NULL
Position	Int	NOT NULL
TopSpeed	Int	NOT NULL
Accleration	Int	NOT NULL
Handling	Int	NOT NULL

Table 4.2: Car Data

Level_track

Name	Data type	Constraints
Level_id	String	PRIMARY KEY
Category	String	NOT NULL
Design	Blob	NOT NULL
Difficulty	Int	NOT NULL

Table 4.3: Level_track Data

Player_Driver

Name	Data type	Constraints
P_name	String	PRIMARY KEY
P_level	Int	NOT NULL
mainScore	Int	NOT NULL
Level_score	Int	NOT NULL

Table 4.4: Player_Driver Data

Game_Environment

Name	Data type	Constraints
E_id	Int	PRIMARY KEY
Weather	String	NOT NULL
Name	String	NOT NULL

Table 4.5: Game_Environment Data

Game

Name	Data type	Constraints
G_id	Int	PRIMARY KEY
gameState	String	NOT NULL
Difficulty	Int	NOT NULL
Options	Int array	NOT NULL

Table 4.6: Game Data

4.3 Diagrams

4.3.1 State Diagram

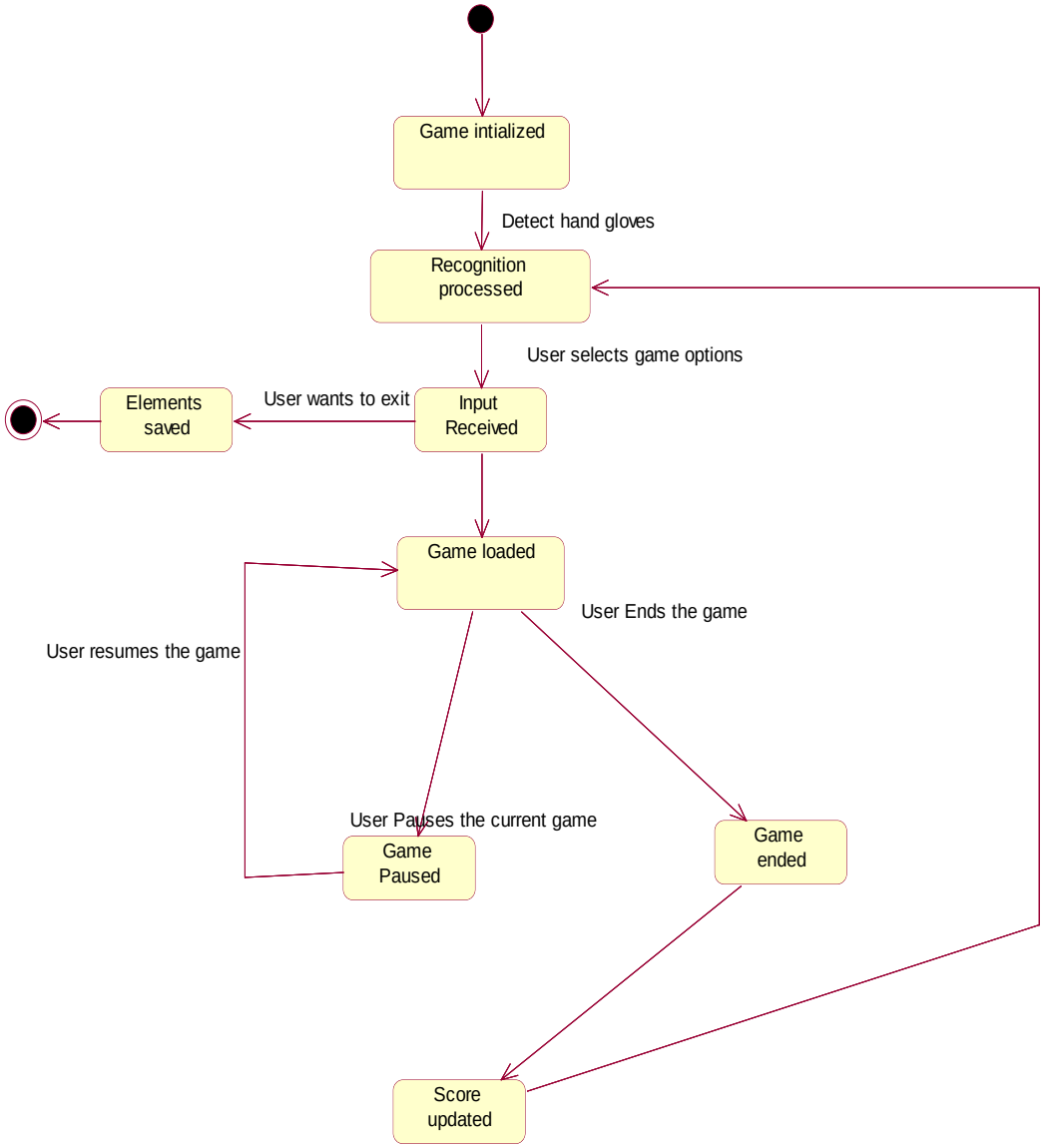

Fig 4.2: State Diagram

4.3.2 Activity Diagram

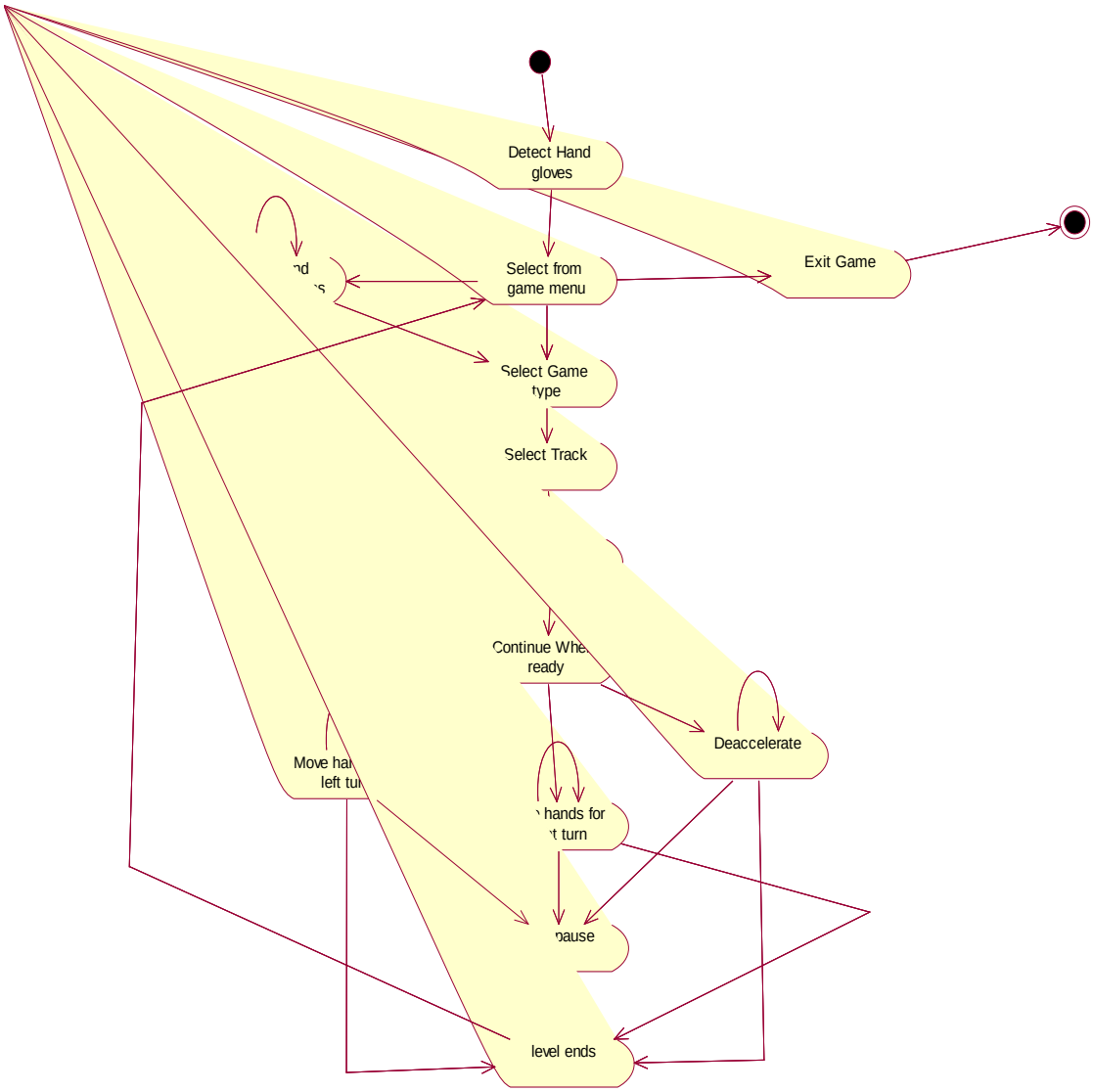

Fig 4.3: Activity Diagram

4.3.3. Class Diagram

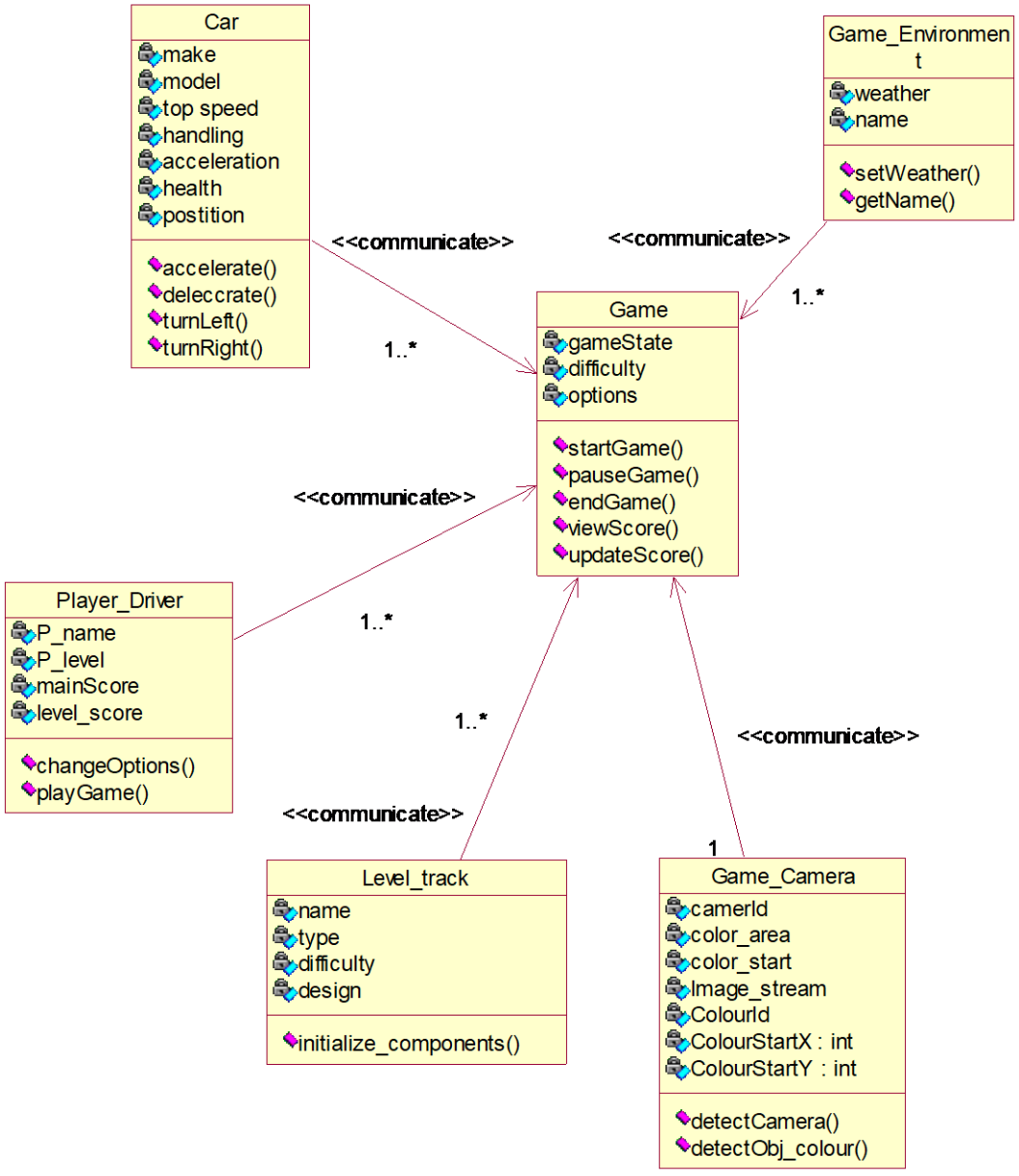

Fig 4.4: Class Diagram

4.4 Functional Modeling

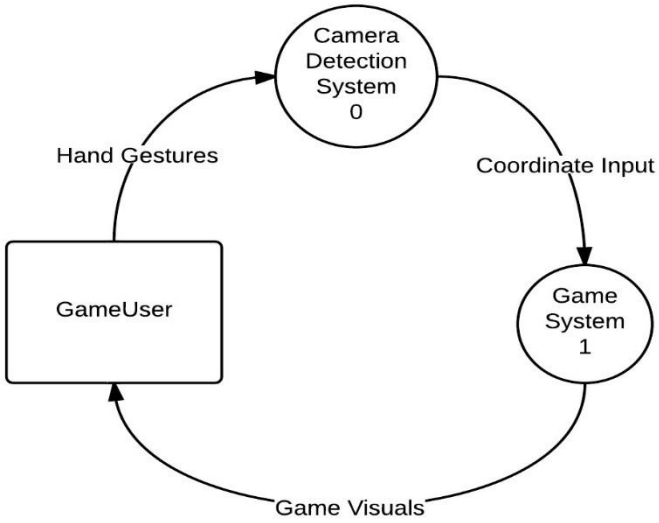

Fig 4.5 Data Flow Diagram Level 0

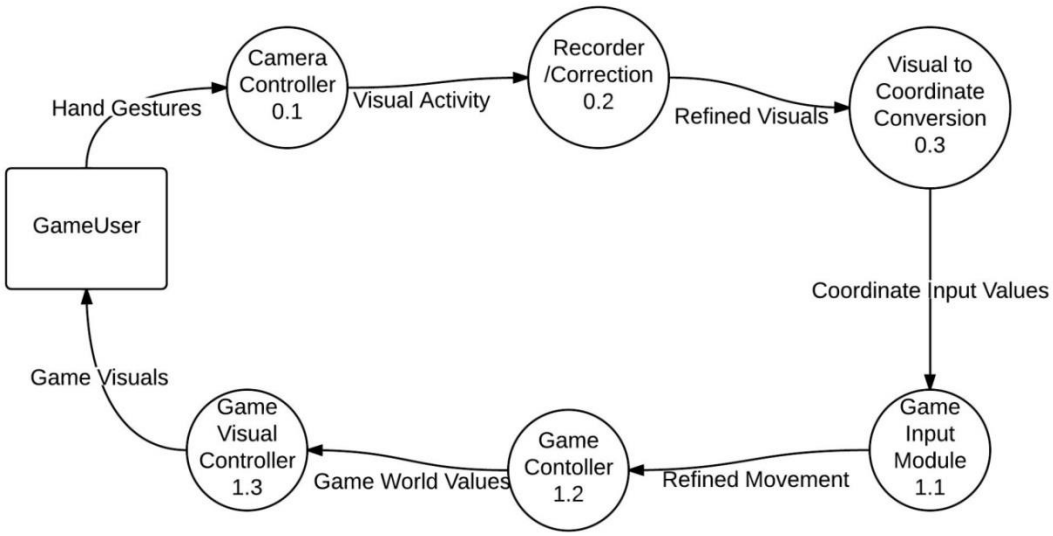

Fig 4.6 Data Flow Diagram Level 1

4.5 TimeLine Chart

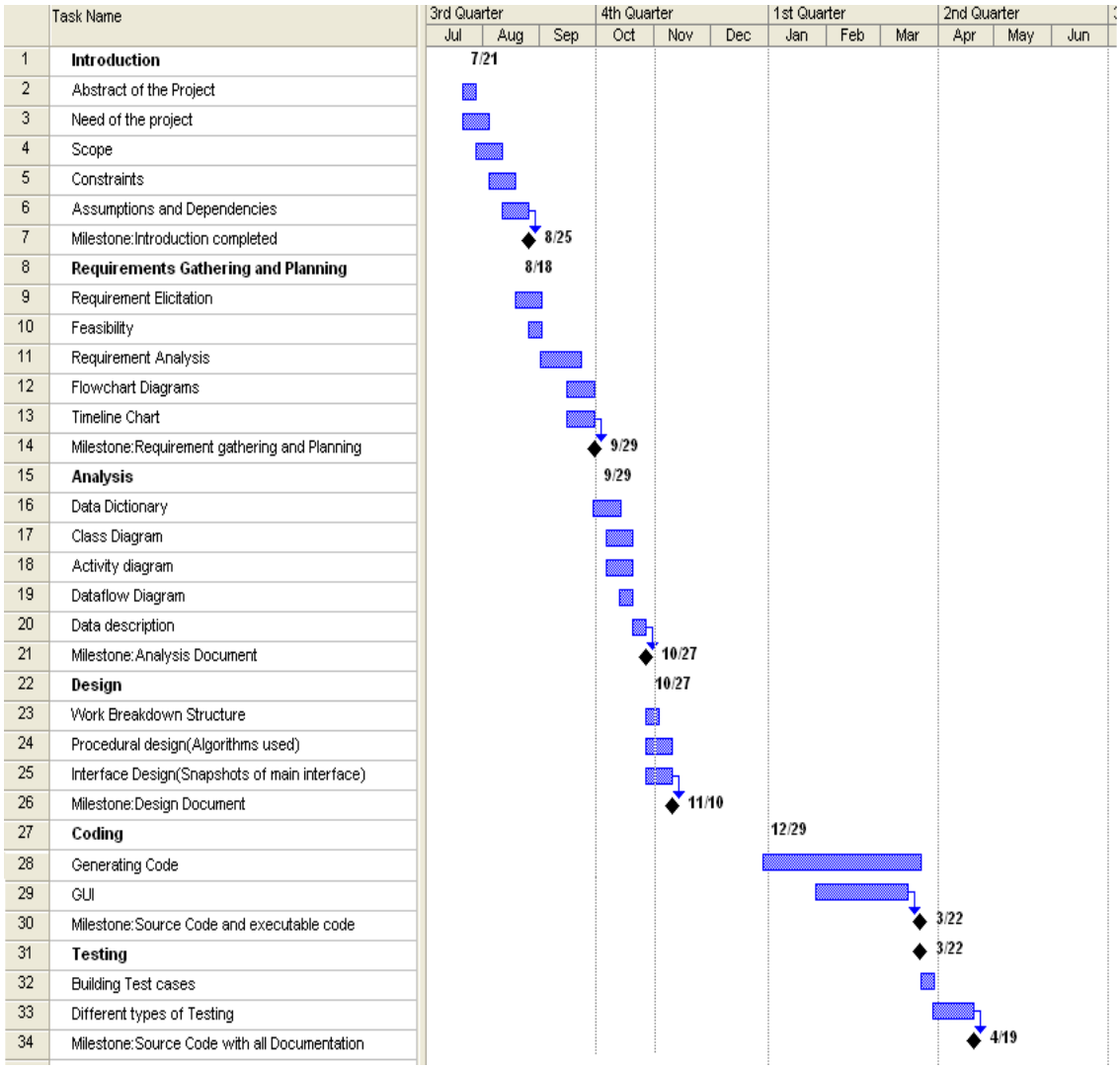

Fig 4.7: TimeLine Chart

5. DESIGN

5.1 Architectural Design

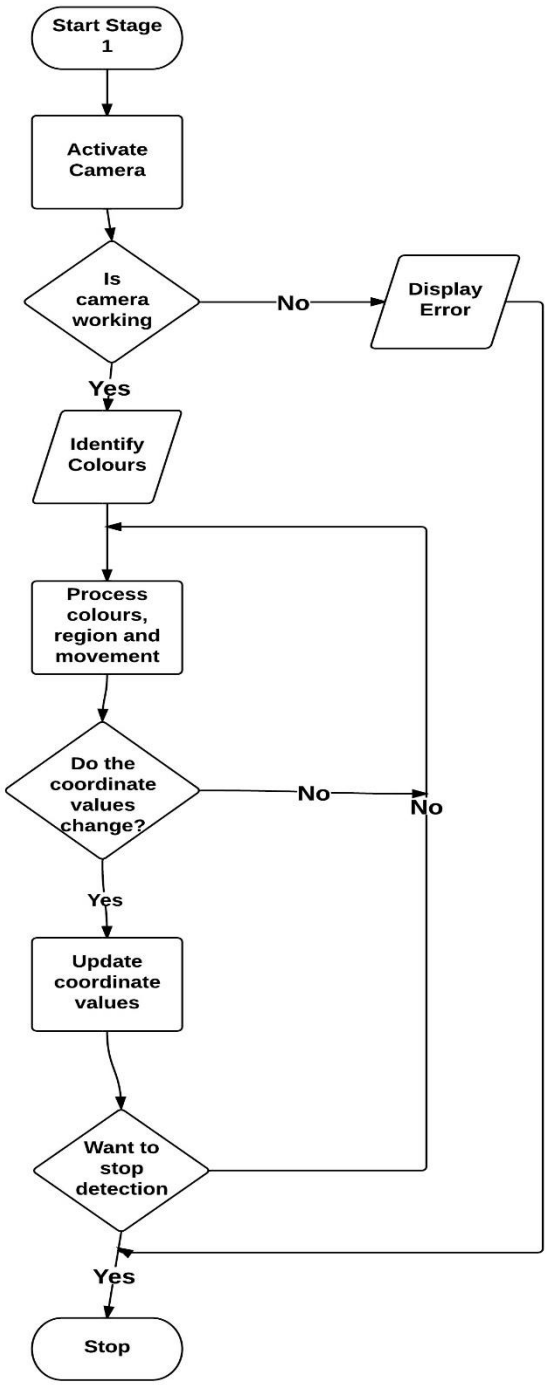

Fig 5.1: Camera Working

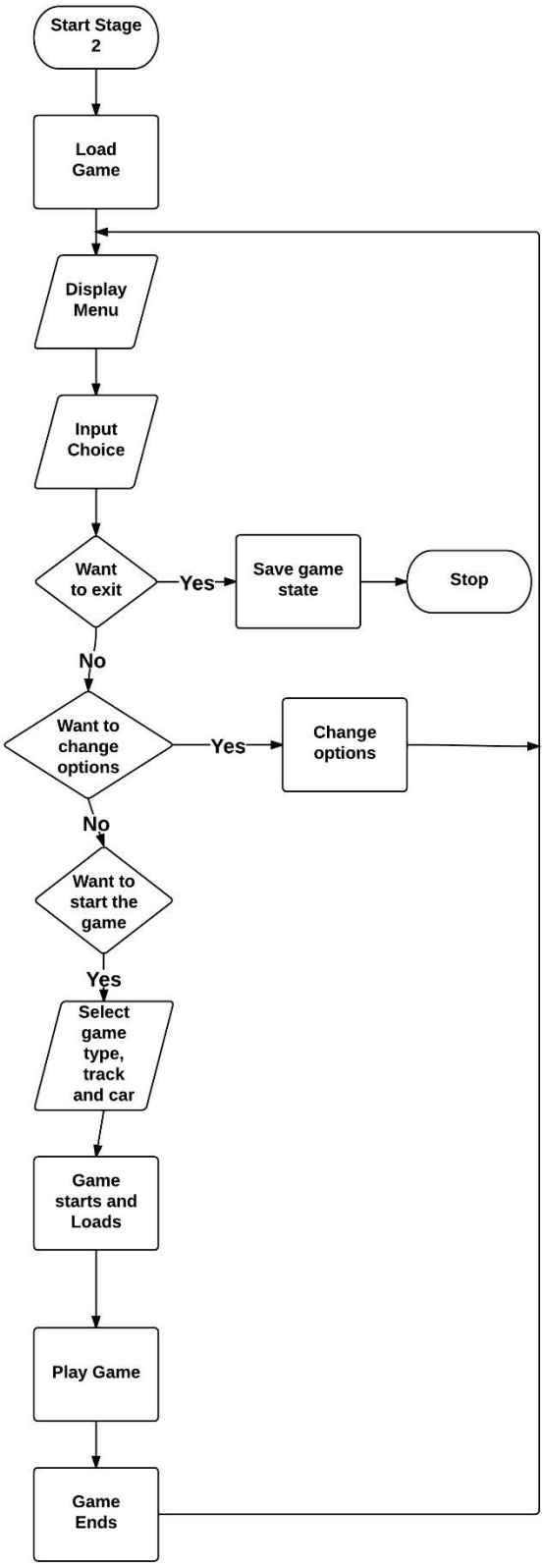

Fig 5.2: System Working

5.1 User Interface Design

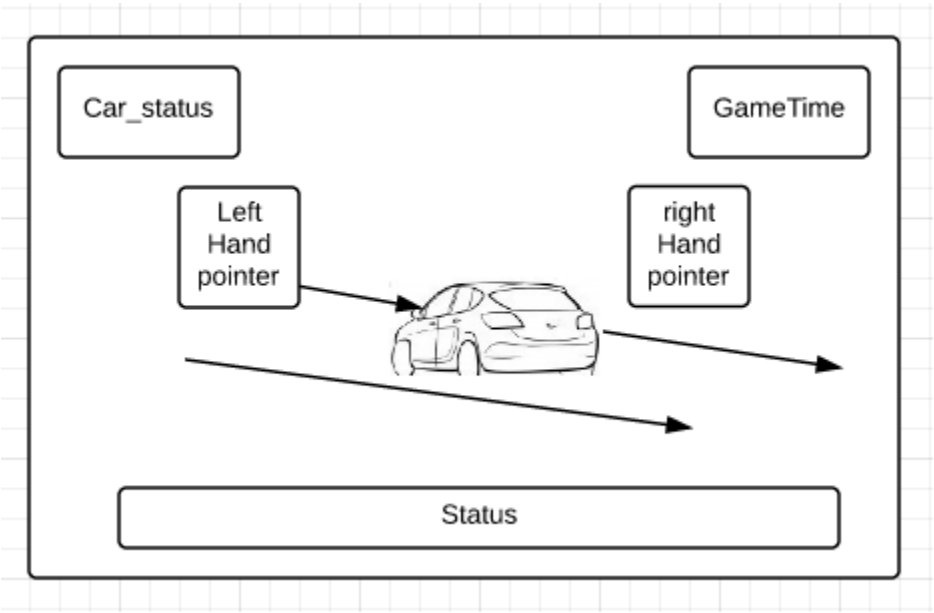

Fig 5.3: Game Interface

6 IMPLEMENTATION

6.1 Hardware and Software Used

6.1.1 Hardware Specification

- Dual-core 2.66-GHz or faster processor
- ATI or Nvidia Graphic Processor
- 256MB Video RAM
- 20 GB HDD
- 2 GB RAM
- Kinect Camera

6.1.2 Software Specification

- Windows XP SP2 or above.
- .NET Framework 4.0
- Compatible Graphics Drivers
- Microsoft Kinect SDK v1.0 or higher

6.2 Algorithms Used

The algorithm is based on tracking game object that are mapped to hands in the system. The variables used are as follows:

- LeftGameObject : game object mapped to left hand
- RightGameObject : game object mapped to right hand
- LeftGameObject.x : x co-ordinate of LeftGameObject in the virtual world system
- LeftGameObject.y : y co-ordinate of LeftGameObject in the virtual world system

Similar holds true for the RightGameObject mapped to the right hand. The threshold_horizontal and threshold_vertical variables stores the experimentally tested threshold values of the minimum distance between LeftGameObject and RightGameObject coordinated values.

If LeftGameObject.y-RightGameObject.y> threshold_vertical

then turn left

Else if LeftGameObject.y-RightGameObject.y<- threshold_vertical

then turn right

Else Go straight

Explanation:

If the y coordinate of LeftGameObject is lesser than that of RightGameObject by threshold difference namely threshold_vertical, that means the user wants the car to turn left. If the y coordinate of LeftGameObject is greater than that of RightGameObject, that means the user wants the car to turn right.

If threshold_horizontal<|LeftGameObject.x-RightGameObject.x|

then Reverse

Else accelerated ahead

Explanation:

If the modulus of the difference between LeftGameObject and RightGameObject is greater than the threshold value namely threshold_horizontal, then user wants the car to stop or reverse back otherwise accelerate forward.

6.3 Working of the project

Car.js

```
private var wheelRadius : float = 0.4;
var suspensionRange : float = 0.1;
var suspensionDamper : float = 50;
var suspensionSpringFront : float = 18500;
var suspensionSpringRear : float = 9000;
var reverseFlag:int=0;
public var brakeLights : Material;
var dragMultiplier : Vector3 = new Vector3(2, 5, 1);
var throttle : float = 0;
private var steer : float = 0;
private var handbrake : boolean = false;

var centerOfMass : Transform;
var frontWheels : Transform[];
var rearWheels : Transform[];
private var wheels : Wheel[];
wheels = new Wheel[frontWheels.Length + rearWheels.Length];
private var wfc : WheelFrictionCurve;
var topSpeed : float = 160;
var numberOfGears : int = 5;
var maximumTurn : int = 15;
var minimumTurn : int = 10;
var resetTime : float = 5.0;
private var resetTimer : float = 0.0;
private var engineForceValues : float[];
private var gearSpeeds : float[];
private var currentGear : int;
private var currentEnginePower : float = 0.0;
```

```
private var handbrakeXDragFactor : float = 0.5;
private var initialDragMultiplierX : float = 10.0;
private var handbrakeTime : float = 0.0;
private var handbrakeTimer : float = 1.0;
private var skidmarks : Skidmarks = null;
private var skidSmoke : ParticleEmitter = null;
var skidmarkTime : float[];
private var sound : SoundController = null;
sound = transform.GetComponent(SoundController);
private var canSteer : boolean;
private var canDrive : boolean;
var countdown = 5;
static var minute = 0;
static var goToMenu : boolean;
static var startTime : float;
static var startRace :float;
static var endTime : float;
static var allowInput : boolean = false;
class Wheel
{
        var collider : WheelCollider;
        var wheelGraphic : Transform;
        var tireGraphic : Transform;
        var driveWheel : boolean = false;
        var steerWheel : boolean = false;
        var lastSkidmark : int = -1;
        var lastEmitPosition : Vector3 = Vector3.zero;
        var lastEmitTime : float = Time.time;
        var wheelVelo : Vector3 = Vector3.zero;
        var groundSpeed : Vector3 = Vector3.zero;
}
```

```
function Start()
{
	accelerationTimer = Time.time;
	startTime = Time.time;
	goToMenu = false;
	SetupWheelColliders();
	SetupCenterOfMass();
	topSpeed = Convert_Miles_Per_Hour_To_Meters_Per_Second(topSpeed);
	SetupGears();
	SetUpSkidmarks();
	initialDragMultiplierX = dragMultiplier.x;
}
function Update()
{
	var relativeVelocity : Vector3 =
transform.InverseTransformDirection(rigidbody.velocity);
	if(Time.time > (startTime+1))
	{
		startTime = Time.time;
		if(--countdown == 0)
		{
			GameObject.Find("g_Countdown").guiText.text = "GO!!!";
			GameObject.Find("Blockade").transform.position.y = 90;
			allowInput = true;
			startRace = 0;
		}
		else if(countdown < 0)
		{
			GameObject.Find("g_Countdown").guiText.enabled = false;
			if(++startRace>59)
			{
```

```
                        startRace=0;
                        minute++;
                }
                GameObject.Find("g_Score").guiText.text = minute+":"+startRace;
        }
        else
                GameObject.Find("g_Countdown").guiText.text = ""+countdown;
}


if(goToMenu)
{
        if(startRace > (endTime + 5))
                Application.LoadLevel(0);

}
if(allowInput)
GetInput();
Check_If_Car_Is_Flipped();
UpdateWheelGraphics(relativeVelocity);
UpdateGear(relativeVelocity);
if(GameObject.Find("g_Checkpoint").guiText.enabled==true && Time.time >
(Checkpoints.delayText + 3))
   GameObject.Find("g_Checkpoint").guiText.enabled = false;
}

function FixedUpdate()
{

        var relativeVelocity : Vector3 =
transform.InverseTransformDirection(rigidbody.velocity);
```

```
        CalculateState();
        UpdateFriction(relativeVelocity);
        UpdateDrag(relativeVelocity);
        CalculateEnginePower(relativeVelocity);
        ApplyThrottle(canDrive, relativeVelocity);
        ApplySteering(canSteer, relativeVelocity);
}



function SetupWheelColliders()
{
        SetupWheelFrictionCurve();
        var wheelCount : int = 0;
        for (var t : Transform in frontWheels)
        {
                wheels[wheelCount] = SetupWheel(t, true);
                wheelCount++;
        }
        for (var t : Transform in rearWheels)
        {
                wheels[wheelCount] = SetupWheel(t, false);
                wheelCount++;
        }
}
function SetupWheelFrictionCurve()
{
        wfc = new WheelFrictionCurve();
        wfc.extremumSlip = 1;
        wfc.extremumValue = 50;
        wfc.asymptoteSlip = 2;
```

```
        wfc.asymptoteValue = 25;
        wfc.stiffness = 1;
}
function SetupWheel(wheelTransform : Transform, isFrontWheel : boolean)
{
        var go : GameObject = new GameObject(wheelTransform.name + " Collider");
        go.transform.position = wheelTransform.position;
        go.transform.parent = transform;
        go.transform.rotation = wheelTransform.rotation;

        var wc : WheelCollider = go.AddComponent(typeof(WheelCollider)) as
WheelCollider;
        wc.suspensionDistance = suspensionRange;
        var js : JointSpring = wc.suspensionSpring;
        if (isFrontWheel)
                js.spring = suspensionSpringFront;
        else
                js.spring = suspensionSpringRear;

        js.damper = suspensionDamper;
        wc.suspensionSpring = js;
        wheel = new Wheel();
        wheel.collider = wc;
        wc.sidewaysFriction = wfc;
        wheel.wheelGraphic = wheelTransform;
        wheel.tireGraphic = wheelTransform.GetComponentsInChildren(Transform)[1];
        wheelRadius = wheel.tireGraphic.renderer.bounds.size.y / 2;
        wheel.collider.radius = wheelRadius;
        if (isFrontWheel)
        {
                wheel.steerWheel = true;
```

```
                go = new GameObject(wheelTransform.name + " Steer Column");
                go.transform.position = wheelTransform.position;
                go.transform.rotation = wheelTransform.rotation;
                go.transform.parent = transform;
                wheelTransform.parent = go.transform;
        }
        else
                wheel.driveWheel = true;

        return wheel;
}
function SetupCenterOfMass()
{
        if(centerOfMass != null)
                rigidbody.centerOfMass = centerOfMass.localPosition;
}

function SetupGears()
{
        engineForceValues = new float[numberOfGears];
        gearSpeeds = new float[numberOfGears];
        var tempTopSpeed : float = topSpeed;
        for(var i = 0; i < numberOfGears; i++)
        {
                if(i > 0)
                        gearSpeeds[i] = tempTopSpeed / 4 + gearSpeeds[i-1];
                else
                        gearSpeeds[i] = tempTopSpeed / 4;
                tempTopSpeed -= tempTopSpeed / 4;
        }
```

```
        var engineFactor : float = topSpeed / gearSpeeds[gearSpeeds.Length - 1];
        for(i = 0; i < numberOfGears; i++)
        {
                var maxLinearDrag : float = gearSpeeds[i] * gearSpeeds[i];// *
dragMultiplier.z;
                engineForceValues[i] = maxLinearDrag * engineFactor;
        }
}

function SetUpSkidmarks()
{
        if(FindObjectOfType(Skidmarks))
        {
                skidmarks = FindObjectOfType(Skidmarks);
                skidSmoke = skidmarks.GetComponentInChildren(ParticleEmitter);
        }
        else
                Debug.Log("No skidmarks object found. Skidmarks will not be drawn");

        skidmarkTime = new float[4];
        for (var f : float in skidmarkTime)
                f = 0.0;
}
function GetInput()
{
        //Kinect Code
        throttle=0;
        if((GameObject.Find("Plane_r").transform.position.y-
GameObject.Find("Plane_l").transform.position.y)>0.1)
        {
```

```
            steer=-0.5;
            print("left");
        }
        else                    if((GameObject.Find("Plane_r").transform.position.y-
GameObject.Find("Plane_l").transform.position.y)<-0.1)
        {
            steer=0.5;
            print("right");
        }
        else
        {
            steer=0;
            print("Go Straight");
        }
        var     left_right=Mathf.Abs(GameObject.Find("Plane_l").transform.position.z-
GameObject.Find("Plane_r").transform.position.z);
        GameObject.Find("GUI2").guiText.text=""+left_right;
        if(left_right>0.5&&left_right<1)
        {
            GameObject.Find("GUI").guiText.text="Reverse ";
            throttle=-1;
        }
        else
        {
            GameObject.Find("GUI").guiText.text="Go";
            throttle=0.3;
        }
        if(throttle < 0.0)
            brakeLights.SetFloat("_Intensity", Mathf.Abs(throttle));
        else
            brakeLights.SetFloat("_Intensity", 0.0);
```

```
	CheckHandbrake();
}

function CheckHandbrake()
{
	if(Input.GetKey("space"))
	{
		if(!handbrake)
		{
			handbrake = true;
			handbrakeTime = Time.time;
			dragMultiplier.x = initialDragMultiplierX * handbrakeXDragFactor;
		}
	}
	else if(handbrake)
	{
		handbrake = false;
		StartCoroutine(StopHandbraking(Mathf.Min(5, Time.time - handbrakeTime)));
	}
}

function StopHandbraking(seconds : float)
{
	var diff : float = initialDragMultiplierX - dragMultiplier.x;
	handbrakeTimer = 1;
	while(dragMultiplier.x < initialDragMultiplierX && !handbrake)
	{
		dragMultiplier.x += diff * (Time.deltaTime / seconds);
```

```
                handbrakeTimer -= Time.deltaTime / seconds;
                yield;
        }

        dragMultiplier.x = initialDragMultiplierX;
        handbrakeTimer = 0;
}

function Check_If_Car_Is_Flipped()
{
        if(transform.localEulerAngles.z > 80 && transform.localEulerAngles.z < 280)
                resetTimer += Time.deltaTime;
        else
                resetTimer = 0;

        if(resetTimer > resetTime)
                FlipCar();
}

function FlipCar()
{
        transform.rotation = Quaternion.LookRotation(transform.forward);
        transform.position += Vector3.up * 0.5;
        rigidbody.velocity = Vector3.zero;
        rigidbody.angularVelocity = Vector3.zero;
        resetTimer = 0;
        currentEnginePower = 0;
}

var wheelCount : float;
function UpdateWheelGraphics(relativeVelocity : Vector3)
```

```
{
	wheelCount = -1;
	for(var w : Wheel in wheels)
	{
		wheelCount++;
		var wheel : WheelCollider = w.collider;
		var wh : WheelHit = new WheelHit();
		if(wheel.GetGroundHit(wh))
		{
			w.wheelGraphic.localPosition = wheel.transform.up * (wheelRadius + wheel.transform.InverseTransformPoint(wh.point).y);
			w.wheelVelo = rigidbody.GetPointVelocity(wh.point);
			w.groundSpeed = w.wheelGraphic.InverseTransformDirection(w.wheelVelo);
			if(skidmarks)
			{
				if(skidmarkTime[wheelCount] < 0.02 && w.lastSkidmark != -1)
				{
					skidmarkTime[wheelCount] += Time.deltaTime;
				}
				else
				{
					var dt : float = skidmarkTime[wheelCount] == 0.0 ? Time.deltaTime : skidmarkTime[wheelCount];
					skidmarkTime[wheelCount] = 0.0;

					var handbrakeSkidding : float = handbrake && w.driveWheel ? w.wheelVelo.magnitude * 0.3 : 0;
					var skidGroundSpeed = Mathf.Abs(w.groundSpeed.x) - 15;
```

```
if(skidGroundSpeed > 0 || handbrakeSkidding > 0)
{
    var staticVel : Vector3 = transform.TransformDirection(skidSmoke.localVelocity) + skidSmoke.worldVelocity;
    if(w.lastSkidmark != -1)
    {
        var emission : float = UnityEngine.Random.Range(skidSmoke.minEmission, skidSmoke.maxEmission);
        var lastParticleCount : float = w.lastEmitTime * emission;
        var currentParticleCount : float = Time.time * emission;
        var noOfParticles : int = Mathf.CeilToInt(currentParticleCount) - Mathf.CeilToInt(lastParticleCount);
        var lastParticle : int = Mathf.CeilToInt(lastParticleCount);

        for(var i = 0; i <= noOfParticles; i++)
        {
            var particleTime : float = Mathf.InverseLerp(lastParticleCount, currentParticleCount, lastParticle + i);
            skidSmoke.Emit(
        Vector3.Lerp(w.lastEmitPosition, wh.point, particleTime) + new Vector3(Random.Range(-0.1, 0.1), Random.Range(-0.1, 0.1), Random.Range(-0.1, 0.1)), staticVel + (w.wheelVelo * 0.05), Random.Range(skidSmoke.minSize, skidSmoke.maxSize) * Mathf.Clamp(skidGroundSpeed * 0.1,0.5,1), Random.Range(skidSmoke.minEnergy, skidSmoke.maxEnergy), Color.white);
        }
    }
    else
    {
```

```
                                        skidSmoke.Emit( wh.point + new Vector3(Random.Range(-0.1, 0.1), Random.Range(-0.1, 0.1), Random.Range(-0.1, 0.1)), staticVel + (w.wheelVelo * 0.05), Random.Range(skidSmoke.minSize, skidSmoke.maxSize) * Mathf.Clamp(skidGroundSpeed * 0.1,0.5,1), Random.Range(skidSmoke.minEnergy, skidSmoke.maxEnergy), Color.white);
                                        }

                                        w.lastEmitPosition = wh.point;
                                        w.lastEmitTime = Time.time;

                                        w.lastSkidmark = skidmarks.AddSkidMark(wh.point + rigidbody.velocity * dt, wh.normal, (skidGroundSpeed * 0.1 + handbrakeSkidding) * Mathf.Clamp01(wh.force / wheel.suspensionSpring.spring), w.lastSkidmark);
                                        sound.Skid(true, Mathf.Clamp01(skidGroundSpeed * 0.1));
                                    }
                                    else
                                    {
                                        w.lastSkidmark = -1;
                                        sound.Skid(false, 0);
                                    }
                                }
                            }
                        }
                        else
                        {
                            w.wheelGraphic.position = wheel.transform.position + (-wheel.transform.up * suspensionRange);
                            if(w.steerWheel)
                                w.wheelVelo *= 0.9;
```

```
                else
                        w.wheelVelo *= 0.9 * (1 - throttle);

                if(skidmarks)
                {
                        w.lastSkidmark = -1;
                        sound.Skid(false, 0);
                }
        }

        if(w.steerWheel)
        {
                var ea : Vector3 = w.wheelGraphic.parent.localEulerAngles;
                ea.y = steer * maximumTurn;
                w.wheelGraphic.parent.localEulerAngles = ea;
                w.tireGraphic.Rotate(Vector3.right * (w.groundSpeed.z /
wheelRadius) * Time.deltaTime * Mathf.Rad2Deg);
        }
        else if(!handbrake && w.driveWheel)
        {
                w.tireGraphic.Rotate(Vector3.right * (w.groundSpeed.z /
wheelRadius) * Time.deltaTime * Mathf.Rad2Deg);
        }
    }
}

function UpdateGear(relativeVelocity : Vector3)
{
    currentGear = 0;
    for(var i = 0; i < numberOfGears - 1; i++)
    {
```

```
            if(relativeVelocity.z > gearSpeeds[i])
                currentGear = i + 1;
        }
}
```

```
function UpdateDrag(relativeVelocity : Vector3)
{
        var relativeDrag : Vector3 = new Vector3(    -relativeVelocity.x                    *
Mathf.Abs(relativeVelocity.x),

        -relativeVelocity.y * Mathf.Abs(relativeVelocity.y),

        -relativeVelocity.z * Mathf.Abs(relativeVelocity.z) );

        var drag = Vector3.Scale(dragMultiplier, relativeDrag);

        if(initialDragMultiplierX > dragMultiplier.x) // Handbrake code
        {
            drag.x /= (relativeVelocity.magnitude / (topSpeed / ( 1 + 2 *
handbrakeXDragFactor ) ) );
            drag.z *= (1 + Mathf.Abs(Vector3.Dot(rigidbody.velocity.normalized,
transform.forward)));
            drag                    +=                    rigidbody.velocity                    *
Mathf.Clamp01(rigidbody.velocity.magnitude / topSpeed);
        }
        else
        {
            drag.x *= topSpeed / relativeVelocity.magnitude;
        }
```

```
        if(Mathf.Abs(relativeVelocity.x) < 5 && !handbrake)
                drag.x = -relativeVelocity.x * dragMultiplier.x;


        rigidbody.AddForce(transform.TransformDirection(drag) * rigidbody.mass *
Time.deltaTime);
}

function UpdateFriction(relativeVelocity : Vector3)
{
        var sqrVel : float = relativeVelocity.x * relativeVelocity.x;
        wfc.extremumValue = Mathf.Clamp(300 - sqrVel, 0, 300);
        wfc.asymptoteValue = Mathf.Clamp(150 - (sqrVel / 2), 0, 150);

        for(var w : Wheel in wheels)
        {
                w.collider.sidewaysFriction = wfc;
                w.collider.forwardFriction = wfc;
        }
}

function CalculateEnginePower(relativeVelocity : Vector3)
{
        if(throttle == 0)
        {
                currentEnginePower -= Time.deltaTime * 200;
        }
        else if( HaveTheSameSign(relativeVelocity.z, throttle) )
        {
```

```
            normPower = (currentEnginePower / engineForceValues[engineForceValues.Length - 1]) * 2;
            currentEnginePower += Time.deltaTime * 200 * EvaluateNormPower(normPower);
        }
        else
        {
            currentEnginePower -= Time.deltaTime * 300;
        }

        if(currentGear == 0)
            currentEnginePower = Mathf.Clamp(currentEnginePower, 0, engineForceValues[0]);
        else
            currentEnginePower = Mathf.Clamp(currentEnginePower, engineForceValues[currentGear - 1], engineForceValues[currentGear]);
}

function CalculateState()
{
        canDrive = false;
        canSteer = false;

        for(var w : Wheel in wheels)
        {
            if(w.collider.isGrounded)
            {
                if(w.steerWheel)
                    canSteer = true;
                if(w.driveWheel)
                    canDrive = true;
```

```
                }
        }
}

function ApplyThrottle(canDrive : boolean, relativeVelocity : Vector3)
{
        if(canDrive)
        {
                var throttleForce : float = 0;
                var brakeForce : float = 0;

                if (HaveTheSameSign(relativeVelocity.z, throttle))
                {
                        if (!handbrake)
                                throttleForce = Mathf.Sign(throttle) * currentEnginePower *
rigidbody.mass;
                }
                else
                        brakeForce = Mathf.Sign(throttle) * engineForceValues[0] *
rigidbody.mass;

                rigidbody.AddForce(transform.forward * Time.deltaTime * (throttleForce
+ brakeForce));
        }
}

function ApplySteering(canSteer : boolean, relativeVelocity : Vector3)
{
        if(canSteer)
        {
```

```
var turnRadius : float = 3.0 / Mathf.Sin((90 - (steer * 30)) * Mathf.Deg2Rad);
var minMaxTurn : float = EvaluateSpeedToTurn(rigidbody.velocity.magnitude);
var turnSpeed : float = Mathf.Clamp(relativeVelocity.z / turnRadius, -minMaxTurn / 10, minMaxTurn / 10);

transform.RotateAround( transform.position + transform.right * turnRadius * steer,
                                                    transform.up,
                                                    turnSpeed * Mathf.Rad2Deg * Time.deltaTime * steer);

var debugStartPoint = transform.position + transform.right * turnRadius * steer;
var debugEndPoint = debugStartPoint + Vector3.up * 5;

Debug.DrawLine(debugStartPoint, debugEndPoint, Color.red);

if(initialDragMultiplierX > dragMultiplier.x)
{
    var rotationDirection : float = Mathf.Sign(steer);
    if(steer == 0)
    {
        if(rigidbody.angularVelocity.y < 1)
            rotationDirection = Random.Range(-1.0, 1.0);
        else
            rotationDirection = rigidbody.angularVelocity.y;
    }
```

```
                transform.RotateAround( transform.TransformPoint( (
        frontWheels[0].localPosition + frontWheels[1].localPosition) * 0.5),

                transform.up,

                rigidbody.velocity.magnitude * Mathf.Clamp01(1 -
rigidbody.velocity.magnitude / topSpeed) * rotationDirection * Time.deltaTime * 2);
            }
        }
}


function Convert_Miles_Per_Hour_To_Meters_Per_Second(value : float) : float
{
        return value * 0.44704;
}

function Convert_Meters_Per_Second_To_Miles_Per_Hour(value : float) : float
{
        return value * 2.23693629;
}

function HaveTheSameSign(first : float, second : float) : boolean
{
        if (Mathf.Sign(first) == Mathf.Sign(second))
            return true;
        else
            return false;
}

function EvaluateSpeedToTurn(speed : float)
```

```
{
	if(speed > topSpeed / 2)
		return minimumTurn;

	var speedIndex : float = 1 - (speed / (topSpeed / 2));
	return minimumTurn + speedIndex * (maximumTurn - minimumTurn);
}

function EvaluateNormPower(normPower : float)
{
	if(normPower < 1)
		return 10 - normPower * 9;
	else
		return 1.9 - normPower * 0.9;
}

function GetGearState()
{
	var relativeVelocity : Vector3 = transform.InverseTransformDirection(rigidbody.velocity);
	var lowLimit : float = (currentGear == 0 ? 0 : gearSpeeds[currentGear-1]);
	return (relativeVelocity.z - lowLimit) / (gearSpeeds[currentGear - lowLimit]) * (1 - currentGear * 0.1) + currentGear * 0.1;
}
```

7 RESULTS AND DISCUSSIONS

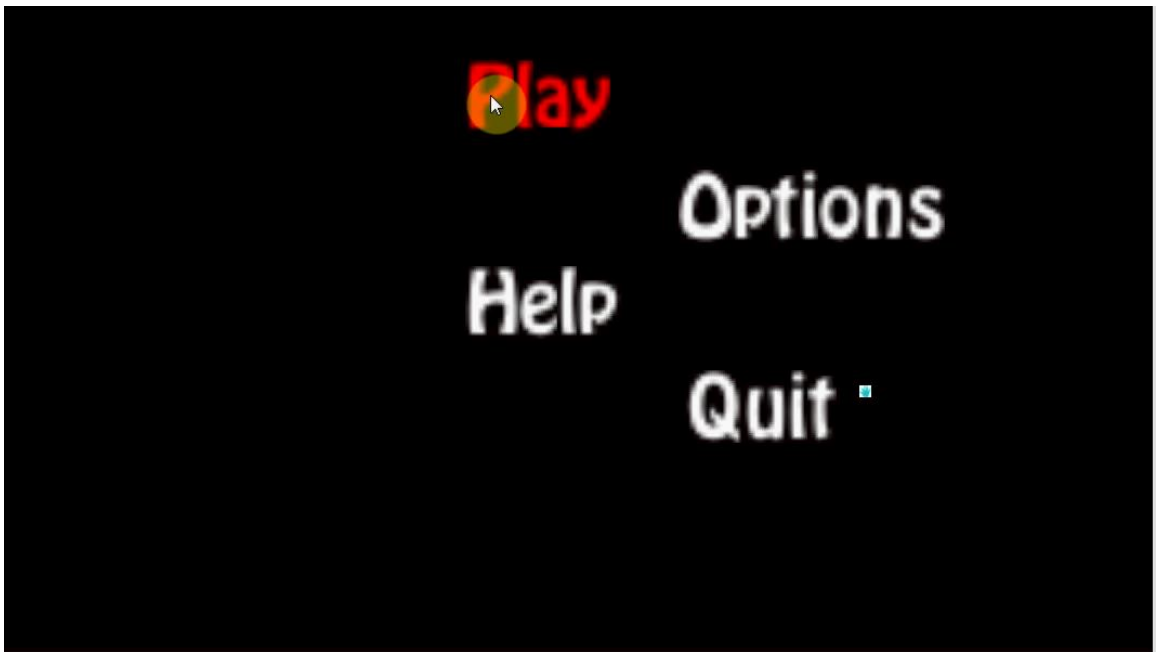

Fig 7.1: Main Menu-Play

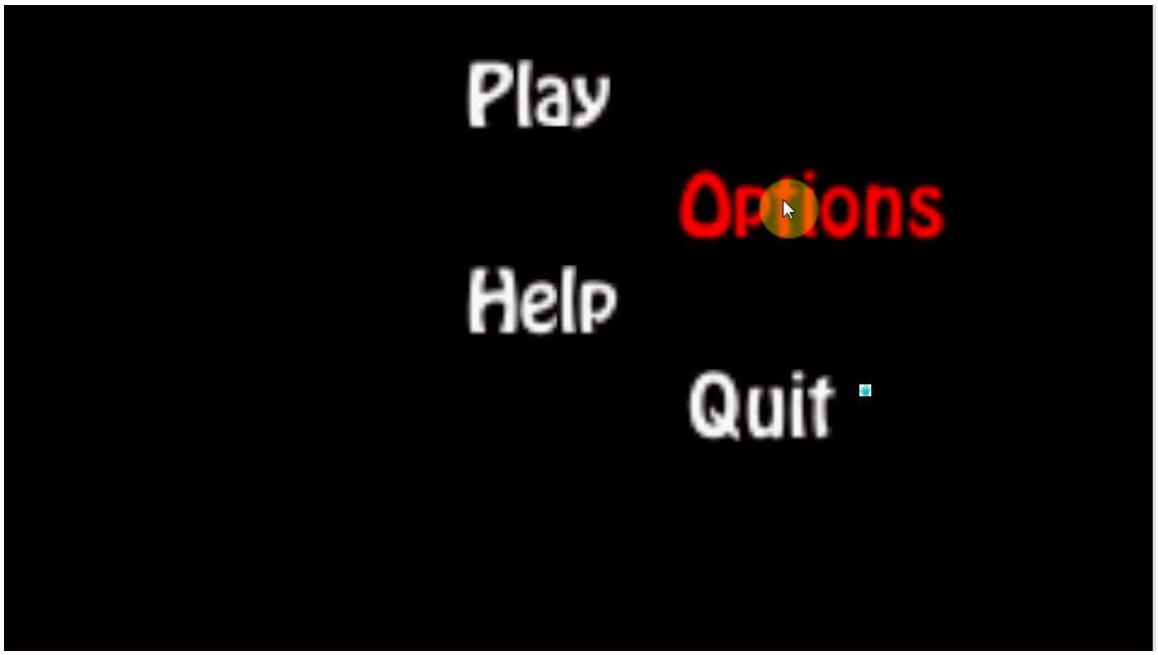

Fig 7.2: Main Menu-Options

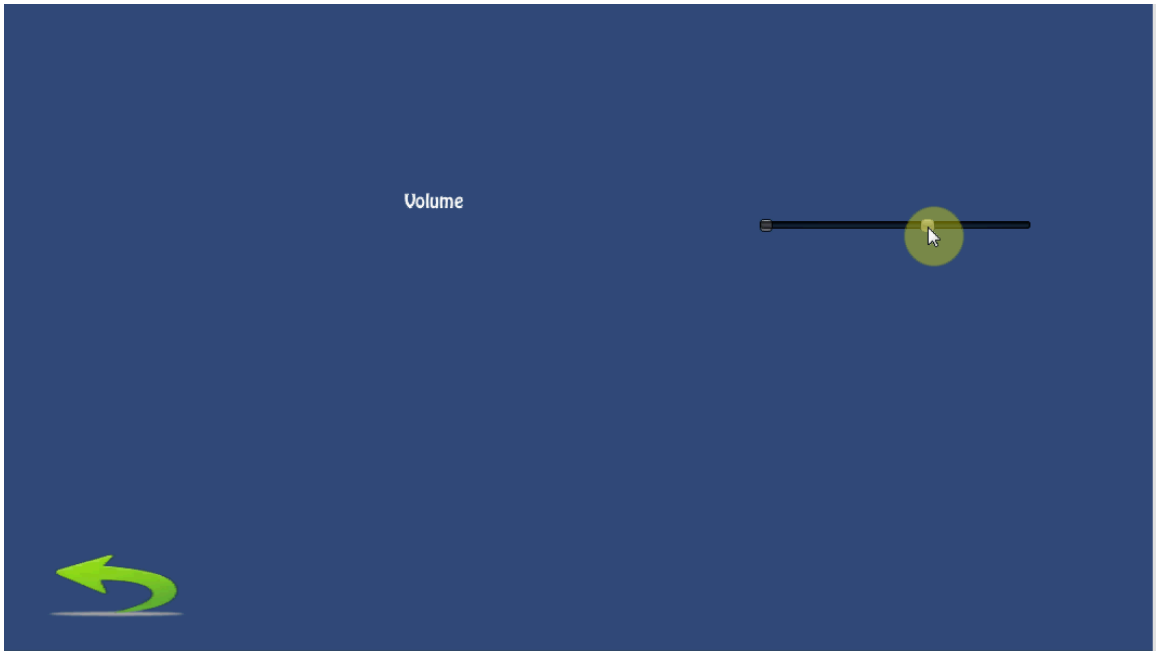

Fig 7.3: Options

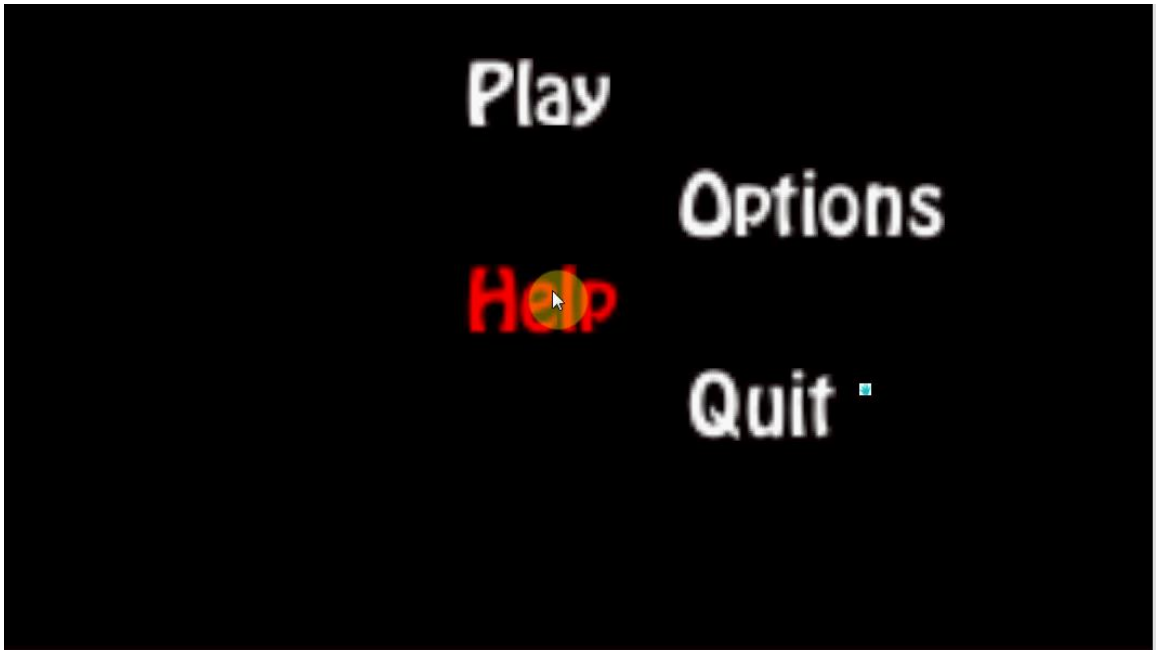

Fig 7.4: Main Menu-Help

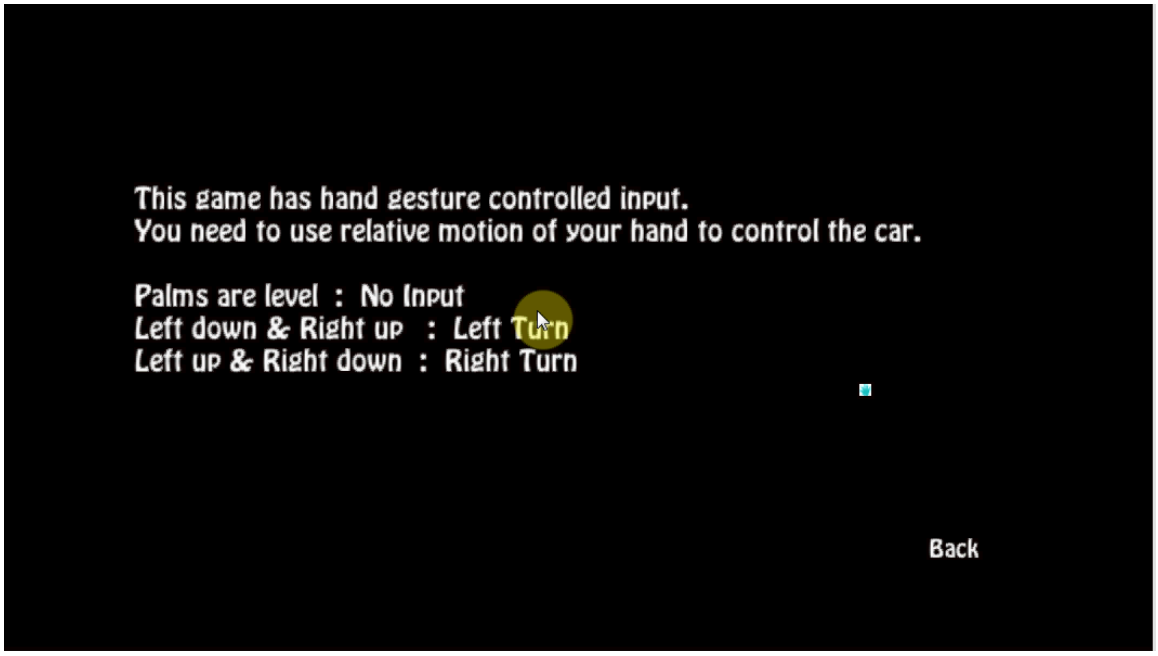

Fig 7.5: Help

Fig 7.6: Play Countdown

Fig 7.7: Game Started

Fig 7.8: Car Left Turn

Fig 7.9: Car Right Turn

Fig 7.10: Car Reverse

Fig 7.11: Car Reverse Left Turn

Fig 7.12: Car Reverse Right Turn

Fig 7.13: Quit

8 CONCLUSION

In this project we have implemented the proposed system successfully using the following technologies:

1. Unity3d Game Engine and MonoDevelop IDE
2. Microsoft Kinect and SDK
3. Kinect Unity Wrapper Package

The implemented system detects and differentiates various body parts and detects hands only relative to the human body and its position relative to the camera. It processes such movements mapping it from real world values to in virtual world values. The virtual world values are further processed that helps the car to move in the virtual world, hence fulfilling the aims of the simulation, that is, the project.

9 FUTURE SCOPE

As any person and object is not perfect, there is a scope of improvement in various aspects of the project. The various ways of possible improvements and enhancements are:

- Voice input processing.
- Enhanced graphics.
- More car models, tracks and races.
- Artificial Intelligence into the system to increase its competitive nature.
- Background story and themes.
- Multiple player detection support.