SAILING SHIP GAME

CE318 HIGH-LEVEL GAME DEVELOPMENT

NAME: YIDING XU

ID: 1402453

TITLE: THE BLACK SAIL

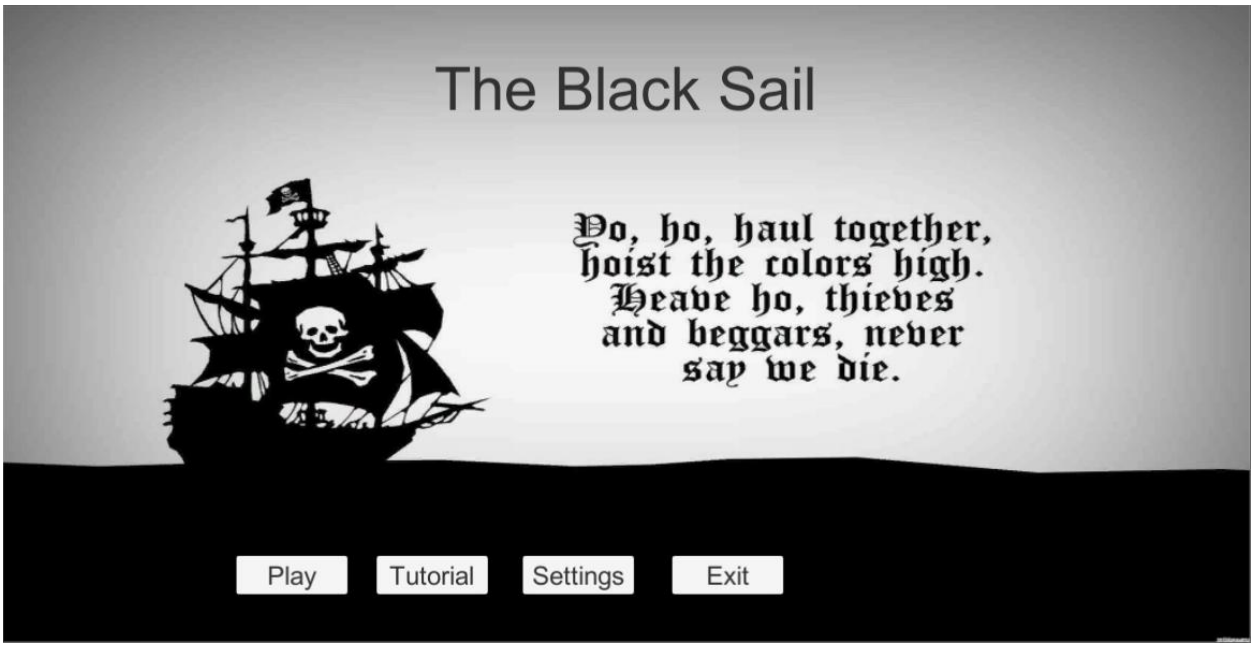


Figure 1 Menu

CONTENTS

PITCH

This game is inspired by my final project which is a 2D turn-based ship combat game based on the age of sail era. I was wondering why not transform it into a real-time 3D combat game, and with the help of Unity5.4 and c# scripts, the imagination finally came true.

The game is all about sailing ships and piracy, where as a player you control a pirate ship exploring the high sea, destroying any ship in your way and collect their loot. The player will find himself relatively small and weak at the beginning, by upgrading the ship he will gradually become increasingly powerful eventually rules.

However, the process is not easy, other pirate ships will attack you actively or laying trap if you want to chase them. Player need to sail the ship intelligently avoiding any unnecessary damage, because repairing the ship cost coins. You want to spend your coin upgrading your ship not repairing.

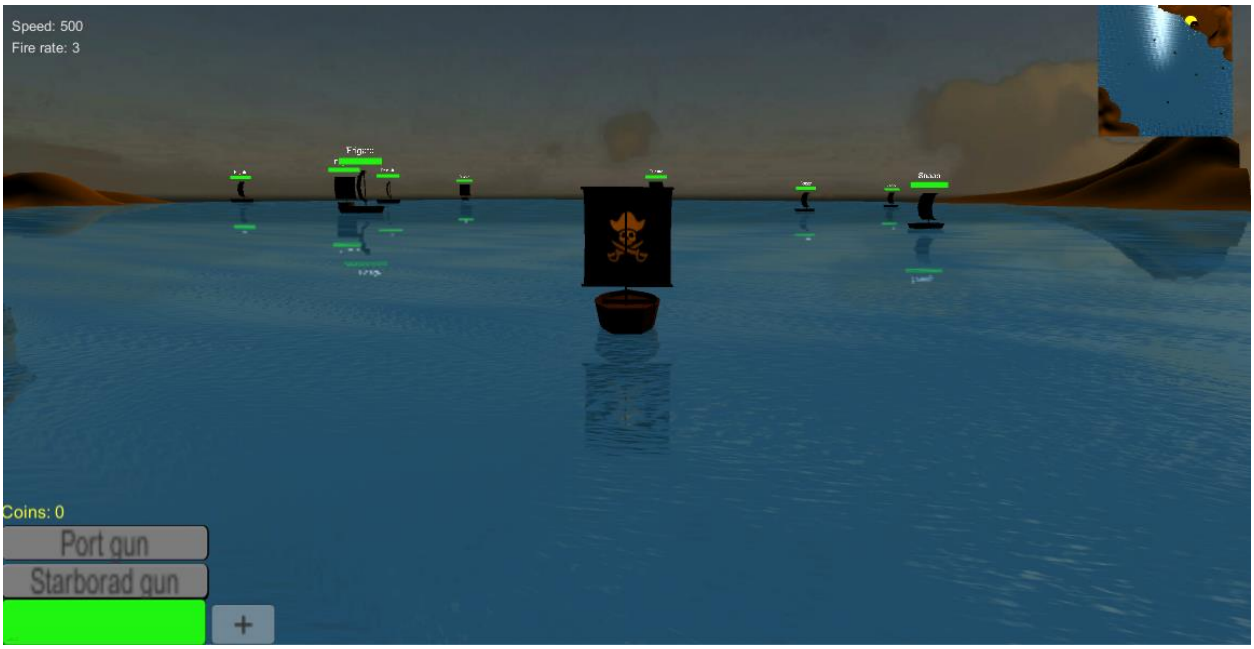


Figure 2 Main Game Scene

LOOK, FEEL AND MUSIC

LOOK

The overall look of this game is very classic, no complex UI or fancy particles, but surrounded by white and black colour with a hint of wooden style. This will help player immerse to the age of sail theme.

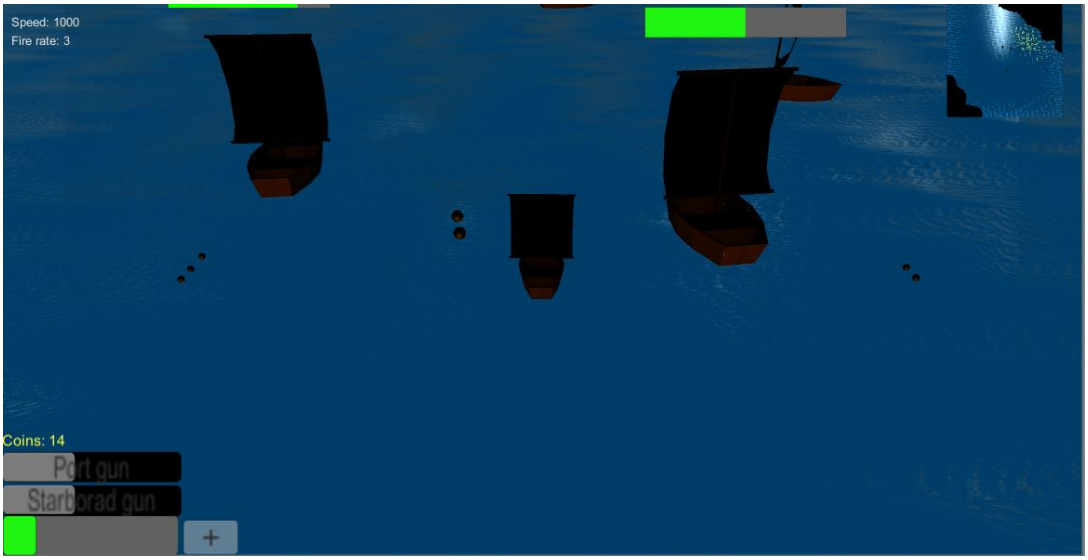


Figure 3 combat with two frigate class

The player can zoom in the camera to receive a more action based battle.

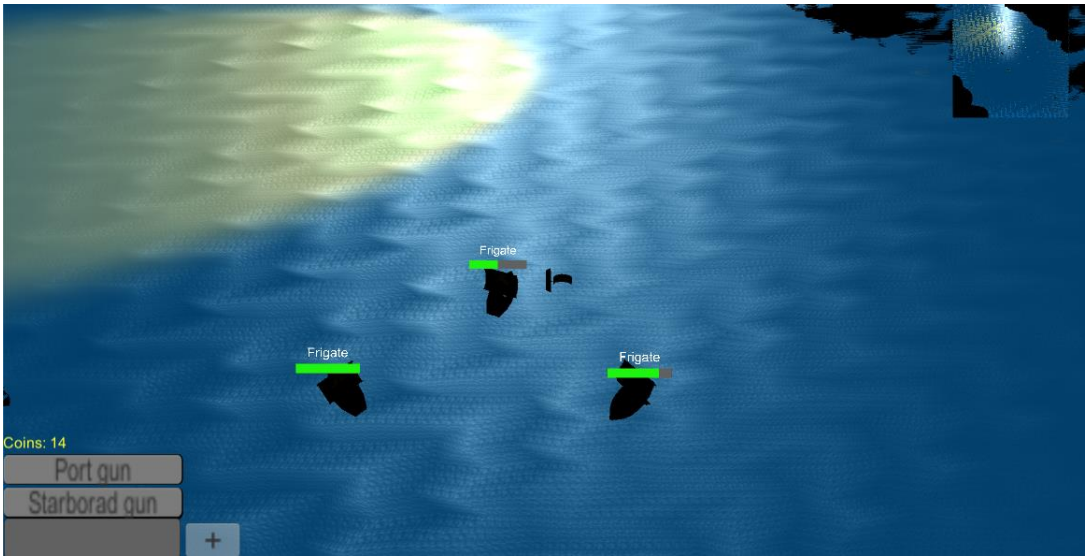


Figure 4 zoomed out camera view

Or a more zoomed out view to have a strategical advantage, unfortunately, I died in that scene as three ships surrounded me.

FEEL

The combat is similar to the sea combat in age of empire 2

Cannon fire from either left side or right side, ship moves and turn slowly.

MUSIC

Music played in the menu is a classic pirate style music. In the game, there is no music playing in the background but a simple sailing sound which is a mix of bell, sea, seagulls, and wood collapsing.

Additionally, sounds will be played when player fire the cannons or the cannon shots impact on objects.

DETAILED DESCRIPTION

PHYSICS

Apart from using colliders which basically is the default unity physics. There is a script called “float” is applied to most of the objects are in the water. It will keep the object floating around the water line. The basic concept of this script is applying an impulse force to the rigid body when it is under the water line.

```
actionPoint = transform.position;
forceFactor = 1f - ((actionPoint.y - waterLevel) / floatHeight);
```

This two line of code will determine how deep is object below the water line. The force factor variable can be negative which means the object is above the water line.

```
if(forceFactor > 0f)
{
    upLift = -Physics.gravity * (forceFactor - rig.velocity.y * bounceDamp);
    rig.AddForce(upLift, ForceMode.Impulse);
}
```

With the force factor found, I apply a force called “uplift” to the object with some adjustable variables such as bounce damp and float height.

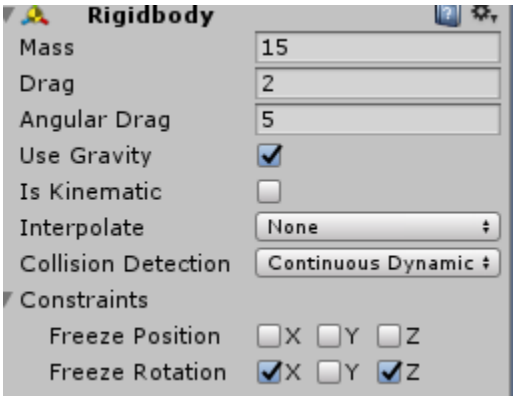
Unfortunately, this is the only extra physic implemented. The buoyancy effect I tried to implement cost too much time and did not have a good result on the model behaviours as it involves many detailed collider modifications.

Therefore, the ships x and z-direction rotation are frozen in the rigid body component.

Back to colliders, all ships using two colliders, one for the hall and one for the sail. A cylinder collider is used for the hall and a box collider for the sail.

```
void OnTriggerEnter(Collider other){
    if (other.gameObject.name == "Terrain"){
        Instantiate (parti, transform.position, transform.rotation);
        Destroy (gameObject);
    }
    if (other.tag == "DummyCollider") {
        Instantiate(parti, transform.position, transform.rotation);
        enemy = other.transform.parent.transform.parent.gameObject;
        enemyHealth = enemy.GetComponent<EnemyHealth>();
        enemyHealth.TakeDamage(10);
        pc.AddCoins(1);
        Destroy(gameObject);
    }
    if(other.tag == "Tower")
    {
        Instantiate(parti, transform.position, transform.rotation);
        Destroy(gameObject);
    }
```

Cannon shots have a sphere collider and set to is a trigger to only collide with a specific object such as ship colliders. The figure above is a part of the script that the cannon shots prefab uses to utilise the trigger collider.



Most game objects have a rigid body and more importantly use gravity to simulate a realistic environment.

Walls are set up around the edge of the map preventing game objects leaving the area, and they also have nav mesh obstacle component for the navmesh agent to use which will be discussed in the later section.

AI

The whole AI implementation is based on a finite state machine(FSM).

A state class holds all available states which AI can have, each state have a transition for it to change state. And a state system class holds a list of states that would be created for an AI. for instance, I have a mediumship controller that handle the AI behaviour for medium ships.

```
public class MediumShipControllor : MonoBehaviour
{
    private GameObject player;
    //public Transform[] path;
    private FSMSystem fsm;

    public void SetTransition(Transition t) {
        Debug.Log(t + " firesd");
        fsm.PerformTransition(t); }

    public void Awake()
    {
        player = GameObject.FindGameObjectWithTag("Player");
    }

    public void Start()
    {
        MakeFSM();
    }
```

This class inherit from MonoBehaviour as it will attach to objects that need this particular behaviour.

In this controller, two states were added wander state and fighting state. Those states are a separate class that inherit from the FSMState class.

```
public class WanderState : FSMState
{
    private NavMeshAgent navAgent;
    private float timer;
    private float wanderTimer = 5.0f;
```

```
public override void Reason(GameObject player, GameObject npc)
{
    if (Vector3.Distance(npc.transform.position, player.transform.position) < 300)
    {
        if (npc.GetComponent<MediumShipControllor>() != null)
        {
            npc.GetComponent<MediumShipControllor>().SetTransition(Transition.SawPlayer);

        }
        if (npc.GetComponent<SmallShipController>() != null){
            npc.GetComponent<SmallShipController>().SetTransition(Transition.SawPlayer);
        }
    }

}

public override void Act(GameObject player, GameObject npc)
{
    //Debug.Log("Wandering" + Vector3.Distance(npc.transform.position, player.transform.position));
    navAgent = npc.GetComponent<NavMeshAgent>();
    npc.GetComponent<Rigidbody>().AddRelativeForce(Vector3.forward * 500);
    timer += Time.deltaTime;

    if (timer >= wanderTimer)
    {
        Vector3 newWayPoint = RandomNavSphere(npc.transform.position, 100000, -1);
        navAgent.SetDestination(newWayPoint);
        timer = 0;
    }
}
```

Each state needs to implement the abstract methods Reason() and Act(). Theses two methods are called in the fixed update.

MEDIUMSHIP AI

As mention above, mediumship has two states wandering and fighting. In wander state which is the initial state. The ship will use a random sphere to pseudo-randomly generate a point in a certain range.

```
private Vector3 RandomNavSphere(Vector3 targetPos, float dist, int layermask)
{
    Vector3 randomDir = UnityEngine.Random.insideUnitSphere * dist;
    randomDir += targetPos;
    NavMeshHit navHit;
    NavMesh.SamplePosition(randomDir, out navHit, dist, layermask);
    return navHit.position;
}
```

This method uses nav mesh to generate a point in the given area and return it as a vector. With the point and a timer the act method can give the ship a wandering behaviour. When “SawPlayer” fired by the system the state will transition into fighting state which the enemy ship will turn broadside on and engage the player.

```
public class FightingState : FSMState {

    public ShipShootingController ssc;

    public FightingState()
    {
        stateID = StateID.Fighting;
    }
```

```
public override void Act(GameObject player, GameObject npc)
{
    ssc = npc.GetComponent<ShipShootingController>();
    npc.GetComponent<Rigidbody>().AddRelativeForce(Vector3.forward * 250);
    npc.transform.rotation = Quaternion.Slerp(npc.transform.rotation, player.transform.rotation, Time.deltaTime / 1);
    Vector3 toPlayer = player.transform.position - npc.transform.position;

    if(Vector3.Angle(toPlayer, npc.transform.right) < 90.0f)
    {
        ssc.fireStarBorad();
    }
    if(Vector3.Angle(toPlayer, -npc.transform.right) < 90.0f)
    {
        ssc.firePort();
    }
}
```

In fighting stage, enemy ship will keep tracking the angle between itself and the player position, by using Quaternion.slerp method enemy can gradually rotate either left side or right side towards the player ship (broadside). Which direction to rotate is decided by comparing the angle from the direction pointing to the player to the direction of the NPC's right vector(or the left vector on the other way) with a fixed shooting angle, in this case, is 90 degree.

When transition LostPlayer is fired the ship will go back to wander stage.

SMALL SHIP AI

Small ship will flee instead of fight when SawPlayer transition is fired.

In fleeing state small ship will turn to the opposite direction of player ship's current velocity and leave explosive barrel behind.

```
public class FleeState : FSMState
{
    private NavMeshAgent navAgent;
    private EnemyHealth eh;
    public GameObject barrel;

    public FleeState()
    {
        stateID = StateID.Flee;
    }
```

```
public override void Reason(GameObject player, GameObject npc)
{

    if (Vector3.Distance(npc.transform.position, player.transform.position) > 300)
    {

        if (npc.GetComponent<MediumShipControllor>() != null)
        {
            npc.GetComponent<MediumShipControllor>().SetTransition(Transition.NoPlayer);
        }
        if (npc.GetComponent<SmallShipController>() != null)
        {
            npc.GetComponent<SmallShipController>().SetTransition(Transition.NoPlayer);
        }
    }
}

public override void Act(GameObject player, GameObject npc)
{
    eh = npc.GetComponent<EnemyHealth>();
    //Debug.Log("Fleeing" + Vector3.Distance(npc.transform.position, player.transform.position));
    navAgent = npc.GetComponent<NavMeshAgent>();
    npc.GetComponent<Rigidbody>().AddRelativeForce(Vector3.forward * 700);
    Vector3 runToPos = Escape(player, npc, 300, -1);
    navAgent.SetDestination(runToPos);
    eh.SpawnBarrel();
}


private Vector3 Escape(GameObject player, GameObject npc, float dist, int layermask)
{
    npc.transform.rotation = Quaternion.LookRotation(npc.transform.position - player.transform.position);
    Vector3 runTo = npc.transform.position + npc.transform.forward * 10;
    NavMeshHit navHit;
    NavMesh.SamplePosition(runTo, out navHit, dist, layermask);
    return navHit.position;
}
```

This method uses nav mesh to find a point of certain distance away in the opposite direction of player ship's current velocity and set it to the current destination.

Unfortunately, the floating physics will not work on those AI as the nav mesh will overwrite the physic applied to the rigid body.

One way to work around it is to only use navmesh to find the path, disable the rotation and position. Use actual physics move objects along the path.

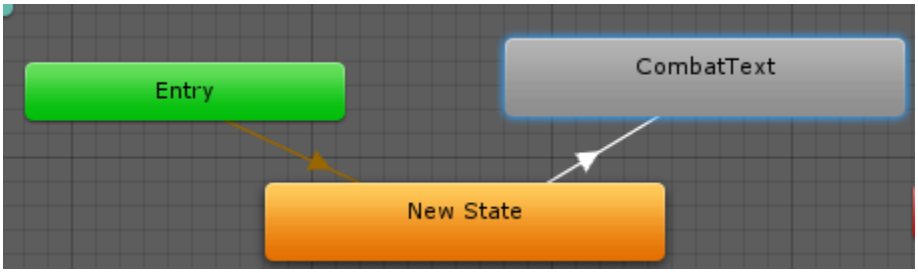
SHIP MODEL

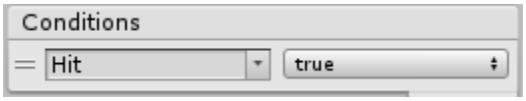
This section will present how the ship was constructed physically.

The ship model itself is a free asset called Lowpoly paper boat, as the model is 90 degree off (the forward vector is the right side of the ship), I create an empty object for the model and rotate it 90 degrees and then put the model object inside the empty object which has the correct rotation. A tail object which locates at the end of the ship which functions as a steering point, the ship will rotate around that point. Two colliders cover the lower boat and the sail which I mentioned in the previous section and two gun group which hold port side gun and starboard side gun respectively.

For the enemy ship, the ship object has an extra canvas object which contains the enemy ship information such as health, name, damage taken. Loot spawn points around the ship which gives the location where the loot will spawn after they die.

A floating combat text system is designed for the enemy ship. An animated text will appear when the ship took damage.





The transition uses a Boolean trigger called “Hit”, whenever ship takes damage the trigger will set to true. The animation is destroyed after 2 seconds.

SERIALIZATION

There is two data file is saved to the default location. First one is the player setting. In the main menu, there is a setting penal which present all the modifiable option, once the player hit the applied button the setting will pass to a JSON string and store in the file. When the game is load the setting will be load at the same time.

```
string jsData = JsonUtility.ToJson(gameSettings, true);
File.WriteAllText(Application.persistentDataPath + "/setting.json", jsData);
```

```
GameSetting gameSettings = JsonUtility.FromJson<GameSetting>(File.ReadAllText(Application.persistentDataPath + "/setting.json"));

musicVolumeSlider.value = gameSettings.musicVolume;
textureQualityDropdown.value = gameSettings.textureQuality;
fullscreenToggle.isOn = gameSettings.fullscreen;
resolutionDropdown.value = gameSettings.resolutionIndex;

resolutionDropdown.RefreshShownValue();
```

```
[System.Serializable]
class GameSetting
{

    public bool fullscreen;
    public int textureQuality;
    public int resolutionIndex;
    public float musicVolume;
}
```

The second file is the player data which include the stat of the player and the current gold of the player. This process is done using binary formatted as the player data is more important, binary file will prevent people easily modify it.

```
public void Save()
{
    BinaryFormatter bf = new BinaryFormatter();
    FileStream file = File.Create(Application.persistentDataPath + "/playerInfo.dat");

    PlayerData data = new PlayerData();
    data.coins = coins;
    data.fireRate = fireRate;
    data.speed = speed;
    data.baseTurnSpeed = baseTurnSpeed;

    bf.Serialize(file, data);
    file.Close();
}
```

```
public void Load()
{
    if(File.Exists(Application.persistentDataPath + "/playerInfo.dat"))
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.Open(Application.persistentDataPath + "/playerInfo.dat", FileMode.Open);
        PlayerData data = (PlayerData)bf.Deserialize(file);
        file.Close();

        coins = data.coins;
        fireRate = data.fireRate;
        speed = data.speed;
        baseTurnSpeed = data.baseTurnSpeed;
    }
}
```

```
[Serializable]
class PlayerData
{
    public int coins;
    public float fireRate;
    public float speed;
    public float baseTurnSpeed;
}
```

ANIMATION

Apart from the floating combat text, there is an animated sun demonstrate a day and night cycle and a spotlight from the tower covering the sea surface.

SOURCE

1. Ship model: Lowpoly paper boat, 3D models Raja Verma, Version 1.0, released on 3rd Map 2016, size 3.3MB.

2. Tower model: Fortress Watchtower, 3D models, Z-craft, Version 1.0, released on 9th April 2014, size 2.3MB.

3. All audio clip come from www.freesound.org.

4. Classic Skybox, Textures & Materials/Skies, mgsvevo, Version1.0, released on 12 November 2014, Size: 4.8 MB

5. Simple Wooden Barrel Pack, 3D Models/Props, Surpent, Version1.2, released on 25 August 2014, Size: 2.2 MB

6. Finite State Machine script, unify community, http://wiki.unity3d.com/index.php?title=Finite_State_Machine, last modified on 10 January 2012, at 21:45.