Ministerul Educaţiei al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare,Informatică și
Microelectronică

Raport

Lucrarea de laborator nr.6 La
disciplina MIDPS

Efectuat: st.gr. TI-143 Besliu Doina
Lutenco Petru

Verificat : Lect.Univ. Cojocaru Svetlana

Chișinău 2016

Tema: Lucru in echipa. Aplicatie complexa

Obiective:

- Crearea unei aplicatii complexe in echipa.
- Divizarea sarcinilor pe membrii echipei **Laboratory Requirements:**

- Lucreaza la proiect in echipa de 2-3 persoane
- Divizeaza task-urile si descrie-le in raport, indicind pentru fiecare cine este responsabil pentru el.
- Inainte de a trece la dezvoltarea proiectului, creeaza o schema cit mai apropiata de rezultatul final *(schema trebuie sa fie primul commit)*
- Fiecare din membrii echipei va lucra pe propriul branch in git, iar una din persoane va avea grija sa faca merge cu master.
- Proiectul se poate afla doar in repozitoriul unui membru al echipei.

- Fiecare din membru va avea propriul raport care va include propriile observatii si concluzii.
    - *Basic Level* (nota 5 || 6):
        -
    - *Normal Level* (nota 7 || 8):
        -
    - *Advanced Level* (nota 9 || 10):
        - Dezvoltarea unei aplicatii:
        - Desktop
        - Mobile
        - Web
        - Browser Extension
        - Game development (web, mobile, desktop)
        - Service application
        - Internet application
        - Client application

      Incercati sa aplicati cit mai multe din thnologiile noi invatate:

- Integrarea cu baza de date
- Folosirea API
- Cross platform
- User friendly

Scripts:

- **PlayerContriller:** *using UnityEngine;*
using System.Collections;

public class PlayerController : MonoBehaviour {

// This component is only enabled for "my player" (i.e. the character belonging to the local client machine).
// This script is responsible for reading input commands from the player
// and then passing that info to NetworkCharacter, which is responsible for
// actually moving things.

NetworkCharacter netChar;

// Use this for initialization
void Start () {
netChar = GetComponent<NetworkCharacter>();
}

// Update is called once per frame
void Update () {
Cursor.visible = false;
// WASD forward/back & left/right movement is stored in "direction"
*netChar.direction = transform.rotation * new Vector3(Input.GetAxis("Horizontal") , 0, Input.GetAxis("Vertical"));*

// This ensures that we don't move faster going diagonally
if(netChar.direction.magnitude > 1f) {
netChar.direction = netChar.direction.normalized;
}

```
		// If we're on the ground and the player wants to jump, set
	// verticalVelocity to a positive number.
	// If you want double-jumping, you'll want some extra code
	// here instead of just checking "cc.isGrounded".
		if(Input.GetButton("Jump")) {
			netChar.isJumping = true;
		}
		else {
			netChar.isJumping = false;
		}

		AdjustAimAngle();

		if(Input.GetButton("Fire1")) {				// Player
wants to shoot...so. Shoot.
	netChar.FireWeapon(Camera.main.transform.position,
Camera.main.transform.forward);
		}

	}

	void AdjustAimAngle() {
		Camera myCamera = this.GetComponentInChildren<Camera>();

		if(myCamera==null) {
			Debug.LogError("Why doesn't my character have a camera?  This is an
FPS!");
			return;
		}

		float aimAngle = 0;

		if(myCamera.transform.rotation.eulerAngles.x <= 90f) {
			// We are looking DOWN
			aimAngle = -myCamera.transform.rotation.eulerAngles.x;
		}
		else {
			aimAngle = 360 - myCamera.transform.rotation.eulerAngles.x;
		}

		netChar.aimAngle = aimAngle;
	}
```

}

- **BotContriller:** *using UnityEngine; using System.Collections; public class BotController : MonoBehaviour {*

// This script controls all aspect of bot AI, including movement and shooting.

// This script is only for "my bot" -- in other words, only the "local" client will have this // enabled. In practice, this means the MASTER client -- which is probably responsible for
// spawning bots.
// REMOTE bots will have this script disabled.

NetworkCharacter netChar;
static Waypoint[] waypoints;

Waypoint destination;
float waypointTargetDistance = 1f;

float aggroRange = 1000000f;

TeamMember myTarget = null; float targettingCooldown = 0;
float targAngleCriteria = 10f; // The angle at which our target needs to be for us to start spraying bullets
float targInnaccuracy = 2f; // Extra innaccuray to simulate mouse hand shake or something

// Use this for initialization
void Start () {
netChar = GetComponent<NetworkCharacter>();

if(waypoints == null) {
waypoints = GameObject.FindObjectsOfType<Waypoint>();
}

destination = GetClosestWaypoint();
}

```
        // Update is called once per frame void
        Update () {
                DoDestination();
                DoDirection();
                DoRotation();

                targettingCooldown -= Time.deltaTime;
                if(targettingCooldown <= 0) {                   DoTargetting();
                        targettingCooldown = 0.5f;
                }

                DoFire();
        }

        Waypoint GetClosestWaypoint() {
Waypoint closest = null;
                float dist = 0;

                foreach(Waypoint w in waypoints) {
                        if(closest==null || Vector3.Distance(transform.position,
w.transform.position) < dist) {
                                closest = w;
                                dist = Vector3.Distance(transform.position,
w.transform.position);
                        }
                }

                return closest;
        }

        void DoDestination() {
if(destination != null) {
                        // We have a destination -- let's check if we have arrived.
                if( Vector3.Distance(destination.transform.position,
transform.position) <= waypointTargetDistance ) {
                                // We have arrived!

                                if(destination.connectWPs != null &&
destination.connectWPs.Length > 0) {
                                        // Pick a connected waypoint
                        destination = destination.connectWPs[
Random.Range(0, destination.connectWPs.Length) ];
```

```
                        }
                        else {
                                // Waypoint isn't connected to anything, which is
kind of a problem.
                                // We need proper navmesh type stuff!
                        }

                }
        }

}
void DoDirection() {
        // We STILL have a destination, so let's move towards it.
        if(destination != null) {
                netChar.direction = destination.transform.position -
transform.position;
                netChar.direction.y = 0;
                netChar.direction.Normalize();

        }
        else {
                // No destination, so let's just stop and be idle.
                netChar.direction = Vector3.zero;
        }

}

void DoTargetting() {
        // Do we have an enemy target in range?
TeamMember closest = null;
float dist = 0;                 foreach(TeamMember tm
in
GameObject.FindObjectsOfType<TeamMember>()) {       // WARNING:
SLOW!                   if(tm == GetComponent<TeamMember>()) {
                // How Zen! We found ourselves.
// Loop to the next possible target!
                        continue;
                }

                if(tm.teamID==0 || tm.teamID !=
GetComponent<TeamMember>().teamID) {
                        // Target is on the enemy team!
```

```
                        float d = Vector3.Distance(tm.transform.position,
transform.position);
                        if( d <= aggroRange ) {
                                // Target is in range!

                                // TODO: Do a raycast to make sure we actually
have line of sight!

                                 // Is the target closer than the last target we found?
                                if(closest==null || d < dist) {
                                        closest = tm;
                                        dist = d;
                                }
                        }
                }
        }

        myTarget = closest;
    } void
    DoRotation() {
        // Let's figure out where we should be facing!
        // By default: Look where we're going.
        Vector3 lookDirection = netChar.direction;

        if(myTarget != null) {
                // We have a target, so let's use that direction as our look
direction!
                 lookDirection = myTarget.transform.position - transform.position;
        }

        // Rotate towards our look direction
        Quaternion lookRotation = Quaternion.LookRotation(lookDirection);
        lookRotation.eulerAngles = new Vector3(0, lookRotation.eulerAngles.y,
0);
        transform.rotation = lookRotation;


        // Now we adjust our aimAngle for animations.
        if(myTarget != null) {
                // Figure out the relative vertical angle to our target and adjust
aimAngle
                Vector3 localLookDirection =
transform.InverseTransformPoint(myTarget.transform.position);
```

```
  float targetAimAngle = Mathf.Atan2(localLookDirection.y, localLookDirection.z) *
Mathf.Rad2Deg;
                        netChar.aimAngle = targetAimAngle;
                }
                else {
                        // We don't have a target, just aim casual
netChar.aimAngle = 0;
                }

        }

        void DoFire() {
                if(myTarget == null)
                        return;

                // Ignore vertical height for determining if we should shoot.
                Vector3 targetPos = myTarget.transform.position;
                targetPos.y = transform.position.y;

                if( Vector3.Angle(transform.forward, targetPos - transform.position ) <
targAngleCriteria ) {


                        // First, get our fire direction in local space
  Vector3 fireDir = Quaternion.Euler(-netChar.aimAngle, 0, 0) * new Vector3(0,0,1);

                        // Add hand shake to make the bot less accurate

                        //Vector3 innaccAngle = new Vector3( Random.Range(-
targInnaccuracy, targInnaccuracy), Random.Range(-targInnaccuracy, targInnaccuracy),
0 );
                        //fireDir = Quaternion.Euler(innaccAngle) * fireDir;

                        //Debug.Log (fireDir);
                        // Convert to global space
                        fireDir = transform.TransformDirection(fireDir);

                        netChar.FireWeapon( transform.position + transform.up * 1.5f ,
fireDir );
                }

        }
```

}

- **NetworkCaracters:**

using UnityEngine;
using System.Collections;

public class NetworkCharacter : Photon.MonoBehaviour {

// This script is responsible for actually moving a character.
// For local character, we read things like "direction" and "isJumping"
// and then affect the character controller.
// For remote characters, we skip that and simply update the raw transform //
position based on info we received over the network.

// NOTE! Only our local character will effectively use this.
// Remove character will just give us absolute positions.
public float speed = 10f; // The speed at which I run
public float jumpSpeed = 6f; // How much power we put into our jump. Change
this to jump higher.

// Bookeeping variables
[System.NonSerialized]
public Vector3 direction = Vector3.zero; // forward/back & left/right
[System.NonSerialized]
public bool isJumping = false;
[System.NonSerialized]
public float aimAngle = 0;

float verticalVelocity = 0; // up/down

Vector3 realPosition = Vector3.zero;
Quaternion realRotation = Quaternion.identity;
float realAimAngle = 0;

Animator anim;

bool gotFirstUpdate = false;

CharacterController cc;

// Shooting Stuff
FXManager fxManager;
WeaponData weaponData;

```
        float cooldown = 0;

        // Use this for initialization
void Start () {
                CacheComponents();
        }

        void CacheComponents() {
if(anim == null) {
                        anim = GetComponent<Animator>();
        if(anim == null) {
   Debug.LogError ("ZOMG, you forgot to put an Animator component on this character
prefab!");
                        }

                        cc = GetComponent<CharacterController>();
                if(cc == null) {
                        Debug.LogError("No character controller!");
                        }

                        fxManager = GameObject.FindObjectOfType<FXManager>();

                        if(fxManager == null) {
                                Debug.LogError("Couldn't find an FXManager.");
                        }
                }

                // Cache more components here if required!
        }

        void Update() {
                cooldown -= Time.deltaTime;
        }

        // FixedUpdate is called once per physics loop
// Do all MOVEMENT and other physics stuff here.
 void FixedUpdate () {   if(
photonView.isMine ) {
                        // Do nothing -- the character motor/input/etc... is moving us
                        DoLocalMovement();
                }
                else {
  transform.position = Vector3.Lerp(transform.position, realPosition, 0.1f);
```

```
  transform.rotation = Quaternion.Lerp(transform.rotation, realRotation, 0.1f);
                        anim.SetFloat("AimAngle",
Mathf.Lerp(anim.GetFloat("AimAngle"), realAimAngle, 0.1f ) );
                }
        }

        void DoLocalMovement () {

                // "direction" is the desired movement direction, based on our player's
input
                Vector3 dist = direction * speed * Time.deltaTime;

                if(isJumping) {
isJumping = false;                      if(cc.isGrounded) {
                verticalVelocity = jumpSpeed;
                        }
                }

                if(cc.isGrounded && verticalVelocity < 0) {
                        // We are currently on the ground and vertical velocity is
                        // not positive (i.e. we are not starting a jump).

                        // Ensure that we aren't playing the jumping animation
                        anim.SetBool("Jumping", false);

  // Set our vertical velocity to *almost* zero. This ensures that:    //   a) We don't start
falling at warp speed if we fall off a cliff (by being close to zero)
                        //   b) cc.isGrounded returns true every frame (by still being
slightly negative, as opposed to zero)
                        verticalVelocity = Physics.gravity.y * Time.deltaTime;
                }
                else {
                        // We are either not grounded, or we have a positive
verticalVelocity (i.e. we ARE starting a jump)

  // To make sure we don't go into the jump animation while walking down a slope, make
sure that
  // verticalVelocity is above some arbitrary threshold before triggering the animation.
                        // 75% of "jumpSpeed" seems like a good safe number, but could
be a standalone public variable too.
                        //
                        // Another option would be to do a raycast down and start the
jump/fall animation whenever we were
```

```
                        // more than ___ distance above the ground.
if(Mathf.Abs(verticalVelocity) > jumpSpeed*0.75f) {
anim.SetBool("Jumping", true);
                        }

                        // Apply gravity.
                        verticalVelocity += Physics.gravity.y * Time.deltaTime;
                }

                // Add our verticalVelocity to our actual movement for this frame
dist.y = verticalVelocity * Time.deltaTime;

                // Adjust our aim angle animation
                anim.SetFloat("AimAngle", aimAngle);

                // Set our animation "Speed" parameter. This will move us from "idle" to
"run" animations,
                // but we could also use this to blend between "walk" and "run" as well.
                anim.SetFloat("Speed", direction.magnitude);

                // Apply the movement to our character controller (which handles
collisions for us)
                cc.Move( dist );
        }



        public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo
info) {
                CacheComponents();

                if(stream.isWriting) {
                        // This is OUR player. We need to send our actual position to the
network.

                        stream.SendNext(transform.position);
                        stream.SendNext(transform.rotation);
                        stream.SendNext(anim.GetFloat("Speed"));
                        stream.SendNext(anim.GetBool("Jumping"));
                        stream.SendNext(anim.GetFloat("AimAngle"));
```

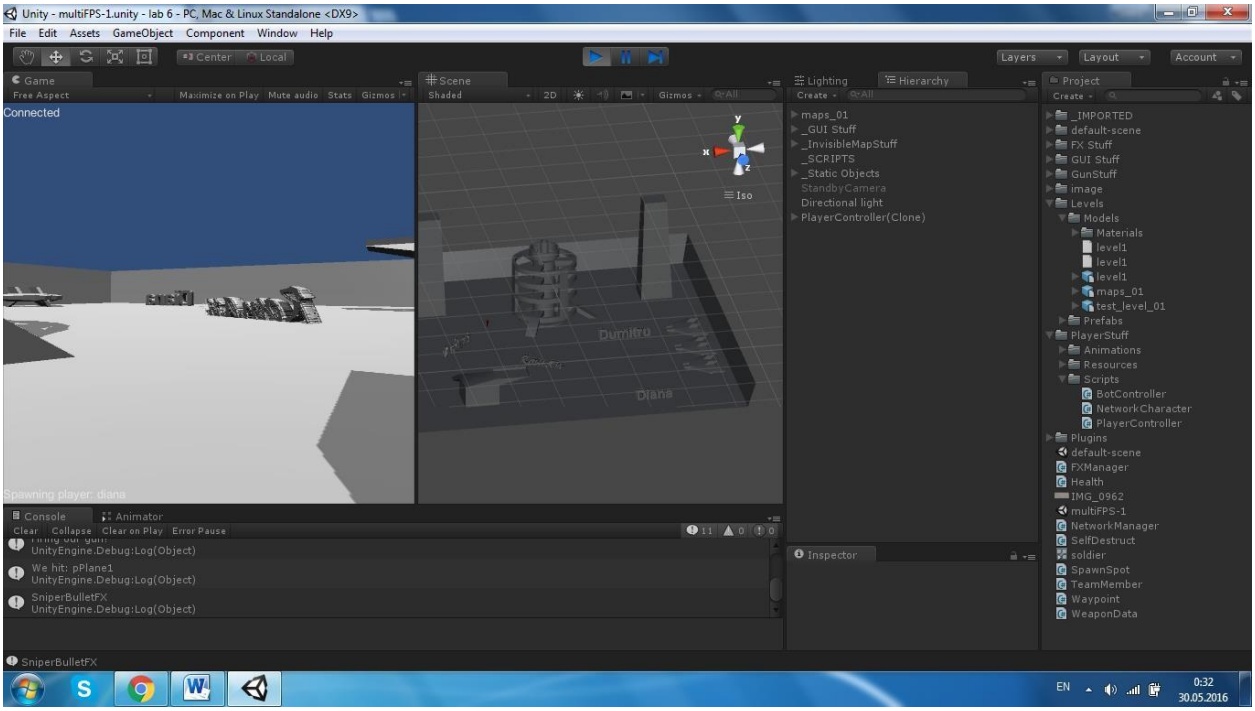
```
                }
                else {
                        // This is someone else's player. We need to receive their position
(as of a few
                        // millisecond ago, and update our version of that player.

                         // Right now, "realPosition" holds the other person's position at the
LAST frame.
                         // Instead of simply updating "realPosition" and continuing to lerp,
                         // we MAY want to set our transform.position to immediately to this
old "realPosition"
                        // and then update realPosition


                        realPosition = (Vector3)stream.ReceiveNext();
realRotation = (Quaternion)stream.ReceiveNext();
anim.SetFloat("Speed", (float)stream.ReceiveNext());
anim.SetBool("Jumping", (bool)stream.ReceiveNext());
realAimAngle = (float)stream.ReceiveNext();

                        if(gotFirstUpdate == false) {
        transform.position = realPosition;
transform.rotation = realRotation;
anim.SetFloat("AimAngle", realAimAngle );
gotFirstUpdate = true;
                        }

                }

        }

        public void FireWeapon(Vector3 orig, Vector3 dir) {
                if(weaponData==null) {
                        weaponData =
gameObject.GetComponentInChildren<WeaponData>();
                        if(weaponData==null) {
                                Debug.LogError("Did not find any WeaponData in our
children!");
                                return;
                        }
                }

                if(cooldown > 0) {
```

```
            return;
        }

        Debug.Log ("Firing our gun!");

        Ray ray = new Ray(orig, dir);
        Transform hitTransform;
        Vector3   hitPoint;

        hitTransform = FindClosestHitObject(ray, out hitPoint);

        if(hitTransform != null) {
            Debug.Log ("We hit: " + hitTransform.name);

            // We could do a special effect at the hit location
            // DoRicochetEffectAt( hitPoint );

            Health h = hitTransform.GetComponent<Health>();

            while(h == null && hitTransform.parent) {
        hitTransform = hitTransform.parent;
                h = hitTransform.GetComponent<Health>();
            }

            // Once we reach here, hitTransform may not be the hitTransform
we started with!

            if(h != null) {
                // This next line is the equivalent of calling:
                //                              h.TakeDamage( damage );
                // Except more "networky"
                PhotonView pv = h.GetComponent<PhotonView>();
                if(pv==null) {
                    Debug.LogError("Freak out!");
                }
                else {

                    TeamMember tm =
hitTransform.GetComponent<TeamMember>();
                    TeamMember myTm =
this.GetComponent<TeamMember>();
```

```
                                        if(tm==null || tm.teamID==0 || myTm==null ||
myTm.teamID==0 || tm.teamID != myTm.teamID ) {
                                                h.GetComponent<PhotonView>().RPC
("TakeDamage", PhotonTargets.AllBuffered, weaponData.damage);
                                        }
                                }

                        }

                        if(fxManager != null) {

                                DoGunFX(hitPoint);
                        }
                }
                else {
                          // We didn't hit anything (except empty space), but let's do a visual
FX anyway
                        if(fxManager != null) {
hitPoint = Camera.main.transform.position +
(Camera.main.transform.forward*100f);
                                DoGunFX(hitPoint);
                        }

                }

                cooldown = weaponData.fireRate;
        }

        void DoGunFX(Vector3 hitPoint) {
                fxManager.GetComponent<PhotonView>().RPC ("SniperBulletFX",
PhotonTargets.All, weaponData.transform.position, hitPoint);
        }

        Transform FindClosestHitObject(Ray ray, out Vector3 hitPoint) {

                RaycastHit[] hits = Physics.RaycastAll(ray);

                Transform closestHit = null;
        float distance = 0;
                hitPoint = Vector3.zero;

                foreach(RaycastHit hit in hits) {
```

```
                    if(hit.transform != this.transform && ( closestHit==null ||
hit.distance < distance ) ) {
                        // We have hit something that is:
                        // a) not us
                        // b) the first thing we hit (that is not us)
                        // c) or, if not b, is at least closer than the previous closest
thing

                        closestHit = hit.transform;
                distance = hit.distance;
                        hitPoint = hit.point;
                    }
                }

                // closestHit is now either still null (i.e. we hit nothing) OR it contains the
closest thing that is a valid thing to hit

                return closestHit;

            }

        }
```

Screenshot:



Concluzie:

In aceasta lucrare de laborator am capatat deprinderi practice în crearea unei aplicatii complexe in echipa și divizarea sarcinilor pe membrii echipei. Am învațat sa creem un joc în Unity. Unity 3D este un engine folosit pentru jocuri 3D, dar cu care se pot creea fara nicio problema si jocurile 2D. Acesta este special pentru faptul ca poate fi folosit de persoane care nu prea au cunostinte de programare.